INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Behavioral Model of Component Frameworks

Sabine Moisan — Annie Ressouche — Jean-Paul Rigault

N° 5065

December 2003

THÈME 3

ISSN 0249-6399 ISRN INRIA/RR--5065--FR+ENG

A Behavioral Model of Component Frameworks

Sabine Moisan [*] , Annie Ressouche [†] , Jean-Paul Rigault [‡]

Thème 3 —Interaction homme-machine,
images, données, connaissances
Projet Orion

Rapport de recherche n° 5065 —December 2003 —52 pages

Abstract: When using a component framework developers need to respect the behavior implemented by the components. Static information about the component interface is not sufficient. Dynamic information such as the description of valid sequences of operations is required. Instead of being in some external documentation, this information should be formally represented and embedded within the components themselves, so that it can be used by automatic tools. We propose a mathematical model and a formal language to describe the knowledge about behavior. We rely on a hierarchical model of deterministic finite state-machines. The communication between the machines follows the Synchronous Paradigm. We favor a structural approach allowing incremental simulation, automatic verification, code generation, and run-time checks. Associated tools may ensure correct and safe reuse of the components. We focus on extension of components through inheritance (in the sense of sub-typing), owing to the notion of behavioral refinement.

Key-words: framework, reusable and adaptable components, behavioral substituability, transition systems, synchronous programming, model checking

[*] INRIA Sophia Antipolis
[†] INRIA Sophia Antipolis
[‡] I3S Laboratory and CNRS

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 —Télécopie : +33 4 92 38 77 65

Modélisation du comportement des composants d'un framework

Résumé : L' utilisation d'un *framework* de composants nécessite le respect du comportement de chacun des composants. Une information statique relative à l'interface des composants n'est pas suffisante pour garantir une bonne utilisation. Une information dynamique comme la description des suites d'opérations valides d'un composant est nécessaire. Cette information, usuellement décrite dans un document annexe, devrait être intégrée dans les composants afin d'être automatiquement analysée par des outils. Nous proposons un modèle mathématique et un langage pour décrire la connaissance relative au comportement des composants. Nous définissons un modèle hiérarchique de machine d'états finis déterministes. La communication entre machines respecte l'hypothèse *synchrone*. Nous adoptons une approche structurelle qui permet simulation incrémentale, vérification automatique, génération de code ainsi que des tests pendant l'exécution. Des outils dédiés permettent une utilisation sure et correcte des composants d'un framework. Le travail présenté dans ce rapport concerne principalement la dérivation de composants (considérée comme une opération de sous-typage), caractérisée par la notion de "raffinement comportemental".

Mots-clés : framework, composants réutilisables et adaptables, substitualité comportementale, systèmes de transitions, programmation synchrone, model checking

1 Introduction

A current trend in Software Engineering is to favor re-usability of code, but also of analysis and design models. This is mandatory to improve product time to market, software quality, maintenance, and to decrease development cost. The notion of frameworks [29] was introduced as a possible answer to these needs.

A framework is dedicated to a family of problems (compiler construction, graphic user interface, knowledge-based systems, etc.). Basically it is a pre-defined architecture composed of generic classes and their relationships. As reusable entities, classes rapidly appeared as too fine grained. Hence, the notion of component frameworks emerged [28]. According to Szyperski [50] a component is "a unit of [software] composition with contractually specified interfaces and explicit context dependencies...". Component-oriented programming now gathers an active community (for instance, see [51, 8]). In the object-oriented approach a component usually corresponds to a collection of interrelated classes and objects providing a logically consistent set of services.

To use a component framework a developer selects, adapts, and assembles components to build a customized application. Thus the major part of the work is *reuse*. Building on re-usability is not straightforward, though. It implies to understand the nature of the contract between the client (i.e., the application developer) and the component. This contract may be the mere specification of a static interface (list of operation signatures), which is clearly not sufficient since it misses any information regarding the component *behavior*. Adding pre- and post-conditions to operations (like in Meyer's *design by contract* [38]) is an interesting improvement. However, the behavior that contracts express is local to an operation. It makes it difficult to comprehend the global valid sequences of operations. The description of such a valid sequence of operations is the major part of what we call the *protocol of use* of the framework. This protocol is often more complex than using a simple library. We claim that expliciting this protocol is an integral part of the components, and that enforcing the respect of the protocol, at compile-time and/or at run-time, should be offered by tools associated with the framework. This report proposes a possible model for embedding the protocol description into components and formal methods to automatically ensure the respect of the protocol.

This work on formalizing component protocols relies on our experience with a framework for knowledge-based system (KBS) engines, named BLOCKS [41, 40]. BLOCKS objective is to help designers create new engines and reuse or modify existing ones, without extensive code rewriting. It consists of around 60 (C++) classes. Apart from support for basic data structures (lists, sets, maps...), most of them are dedicated to knowledge representation artifacts such as the classical AI notions of *frame* and of *rule* [19]. The methods of BLOCKS classes are used by the framework user to construct new knowledge-based system engines. Each class comes with a behavioral description of the valid sequences of operations, in the form of state-transition diagrams. Such a description allowed us to prove invariant properties of the framework, using model-checking techniques. As with other frameworks, the developer adapts BLOCKS classes essentially through subtyping (more exactly, class derivation used as subtyping). The least that can be expected is that the derived classes respect the behavioral protocol that the base classes implement and guarantee. In particular, we want to ensure that an invariant property at the framework base level also holds at the developer's class level. Thus the notion of *behavioral substitutability* is central to such a safe use of the framework.

To this end we chose to elaborate a formal model of behavioral substitutability so that we may lay design rules on top of it. In this model safety properties are preserved during subtyping. Our aim is to propose a verification algorithm as well as practical design rules to ensure sound framework adaptation.

2 Target Framework Characteristics

2.1 Notion of Components

In the object-oriented community a component framework [7, 28] is usually composed of hierarchies of classes that the developer may compose or extend. The root class of each hierarchy corresponds to an important concept in the target domain. In this context, a component can be viewed as the realization of a sub-tree of the class hierarchy: this complies to one of Szyperski's definitions for components [50].

As a matter of example, let us examine the problem of history management in an object-oriented environment. In our framework (BLOCKS) an *history* is composed of several successive *snapshots*, each one gathering the modifications (or *deltas*) to object attributes that have happened since the previous snapshot (that is during an execution step). It seems to be a rather general view of history management and any framework with a similar purpose is likely to provide classes such as `History`, `Snapshot` and `Delta`, as shown in the UML [47] class diagram of figure 1. Class `Snapshot` memorizes the modifications of objects during an execution step in its attached `Delta` set; it displays several operations: memorize the deltas and other contextual information, check the validity of some condition at this given step in the execution, add a new delta, and add a child snapshot (i.e., close the current step and start a new one). Calls to these operations will appear in the code of other operations (generally located in other classes). For instance, a traversal operation on past snapshots in the `History` class may demand to check conditions on snapshots.

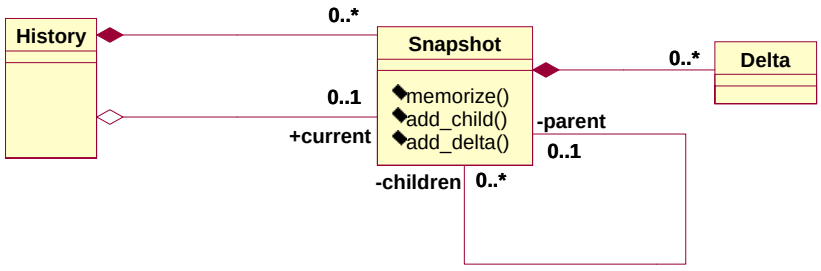


Figure 1: Simplified UML diagram of class `Snapshot`.

2.2 Using a Framework

Framework users both adapt the components and write some glue code. They will (non-exclusively) use these components directly (like a library), or extend the classes they contain by inheritance, or compose the classes together, or instantiate new classes from predefined generic[1] ones. Among all these possibilities, class derivation is very frequent. It is also the one that may raise the trickiest problems. In the sequel we shall mainly concentrate on it.

[1] class templates in C++

When deriving a class the user may either introduce new attributes and/or operations or redefine inherited operations. These extensions should be “semantically acceptable”, i.e., they should comply to the design hypotheses of the framework, i.e., respect the framework invariants.
Let us continue with our example: the `Snapshot` class originally does not take into account a possible “backtrack” (the “linear” history of `Snapshot` becomes a “branching” one). However this possibility is necessary in simulation activities: a common practice is to backtrack to past milestones; the aim is to try a different action or to modify some contextual information and see what happens. To cope with such requirements, the user can introduce a `BacktrackableSnapshot` class as a derivative of `Snapshot` (figure 2). In this particular example, the inherited operations need no redefinition[2]; this class just defines two operations: `regenerate`, that reestablishes the memorized values and `search` that checks whether a condition was true in a previous state. The regeneration feature implies that deltas have the ability to redo and undo their changes; hence the new class `BacktrackableDelta` has to be substituted to `Delta`. Relying on the static information of the class diagram of `Snapshot` (signatures of methods and associations among classes), the framework user obtains the inheritance graph shown on figure 2.

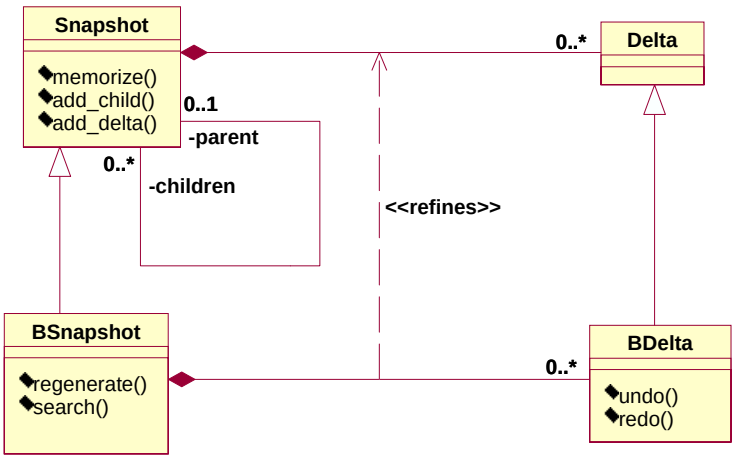


Figure 2: UML class diagram of `BacktrackableSnapshot`; above, the original classes, below, the derived ones.

2.3 Protocol(s) of Use

Static information is not sufficient to ensure a safe and correct use of the framework: specifying a *protocol of use* is required. This protocol is defined by two sets of constraints. First, a *static* set enforces the internal consistency of class structures:

[2] In the general case, there would be new operations as well as redefined operations. Our approach is able to cope with both cases.

- UML-like class diagrams provide a part of this information: input interfaces of classes (list of operation signatures), specializations, associations, indication of operation redefinitions, etc.
- But class diagrams do not display any constraint on the operations that a component expects from other components (a sort of *output interface*, something something that exists in e.g., UML-RT [49] and that will likely find its way into UML 2.0);
- Nor can class diagrams easily express static properties specific to the implementation; for instance, in C++, class derivation and composition demand a scaffolding of structure-dependent construction/destruction operations.

We do not focus on this part of the protocol since its static nature makes it easy to generate the necessary information at compile-time.
A second set of constraints describes *dynamic* requirements:

1. Legal sequences of operation calls;
2. Specification of internal behavior of operations and of the sequence of messages these operations send to other components;
3. Specification of the valid ways to redefine operation behavior in derived classes.

These dynamic aspects are more complicated to express than static ones, they are error-prone, and there is no tool (as natural as compiler-like tools for the static case) to handle and check them. While item 1 (and partially item 2) can be addressed by classical UML state-transition models (Statecharts), the complete treatment of the last two items is more challenging. We shall propose a solution in section 3.

2.4 Implementation of the Protocol of Use

To implement the protocol several non-exclusive Software Engineering techniques have been proposed. The most popular ones include:

- Using well-defined design patterns [20]; this makes it possible, e.g., to create polymorphic objects (abstract factory, virtual constructor, singleton, prototype), to traverse complex data structures (iterator, visitor), and to implement polymorphic algorithms (strategy). This helps clarify the software architecture, but it seldom is a complete solution.
- Meta-programming [31] or more generally any form of *generative programming* [17]; indeed these are powerful tools to automate protocols of use and behavioral refinement; they are well adapted to static protocol issues. They allows to generate the code to implement the protocol. In our case we use the OpenC++ meta-object protocol [11, 13] to implement, for instance, some specific “aspects” [32] of classes such as introspection or persistence. It also helps generate the language-dependent “scaffolding” of constructors requested by class derivation. However, meta-programming is complex. Further, the knowledge about components is located in a separate (meta) program, external to the components, a risk of inconsistent evolution.

- Embedding the knowledge for using, deriving, and composing within the component itself. A behavioral modeling with associated proofs and simulations allows compile-time as well as run-time verifications relying on this knowledge.

The context of our work is related to the third item. In this report we concentrate on proposing models to formally describe component behavior. The precise way to embed this information into physical components is an implementation dependent issue, not developed here.
Although some works already address the dynamic behavior specification of components [46, 48], there is no complete and consensual technique for representing and embedding the corresponding information. For instance, in JavaBeans [42], the embedded knowledge is static and rather poor; in classical CORBA, the IDL is external to the components and is not much richer; in the new CORBA Component Model [37, 18] the description remains static. In the next section we present a possible solution to enrich the description with dynamic behavior information.

3 Behavior Description and Behavior Refinement

Our approach is threefold. First, we define a mathematical model providing consistent description of *behavioral entities*. In the model, behavioral entities are whole components, sub-components, single operations, or any assembly of these. Hence, the whole system is a hierarchical composition of communicating behavioral entities. Such a model complements the UML approach and allows to specify the class and operation behavior with respect to class derivation. Second, we propose a *hierarchical* behavioral specification *language* to describe the dynamic aspect of components. In the third place, we define a *semantic* mapping to bridge the gap between the specification language and its meaning in the mathematical model.
As already mentioned, our primary intent is to formalize the behavior side of class derivation, in the sense of *subtyping*[3]. In the object-oriented approach, subtyping usually obeys the classical Substitutability Principle [33]. This principle has a static interpretation which leads to, for instance, the well-known covariant and contravariant issues for parameters and return types. But it may also be given a dynamic interpretation, leading to behavioral subtyping, or *behavioral substitutability* [23]. This is the kind of interpretation we need to enforce the dynamic aspect of framework protocols, since it provides a notion of behavior-wise safe derivation.
We focus on proving specifications, not implementations. Although we use concurrency for our specification, modeling a system as disjoint parts with (more or less) independent execution, it is only a *logical* notion, not an implementation one. This work does not address concurrent execution models.
To deal with behavioral substitutability, we need behavior representation formalisms: we propose to rely on the family of *synchronous* models [5, 21]. These models are dedicated to specify event-driven and discrete time systems. Such systems interact with their environment, reacting to input events by sending output events. Furthermore, they obey the *synchrony hypothesis*: the corresponding reaction is *atomic*; during a reaction, the input events are frozen, all events are considered as *simultaneous*, events are *broadcast* and available to any part of the system that listens to them. As a consequence, output events are simultaneous to the input events which raise them. A reaction is also called an *instant*. The succession of instants defines a logical time. The major interest of synchronous models is that their verification exhibits a lower computational complexity than asynchronous ones, which is the main reason for our choice. Moreover, to describe complex behavior as found in component protocols of use, a hierarchical modular description is natural. Provided that certain "compositionality properties" hold, automatic proofs become modular and thus more efficient.

3.1 Behavior Description Language

The behavorial description language (called BDL) that we propose belongs to the family of *synchronous* languages. Such languages are devoted to specify *reactive systems* [24] : systems which interact continuously with their environment, receiving input events and sending output events. *Determinism* is an important characteristic of reactive programs. A deterministic reactive program pro-

[3]Note that, in this paper, derivation, inheritance, specialization all refer to the *subtyping* interpretation. In particular, we do not consider the other uses or interpretations of inheritance that some programming languages may offer.

duces always the same output events when fed with identical input event sequences. Synchronous languages have a mathematical semantics based on the *synchrony hypothesis* Synchronous languages are dedicated to sequential programming. Parallel composition exists as an operator, but does not imply a concurrent execution (threads, sockets...). Synchronous programming fits the behavior component specification. Moreover, it leads to mathematical models more convenient to verify than asynchronous ones and the language operators are easily defined.
The primitive elements of the language are the *automata*. Automaton structure is well-suited to describe event-driven systems, running in parallel and using hierarchical decomposition. An automaton is composed of a finite set of states and a transition relation. The transitions are labeled by a conditional part and an action part. Conditions and actions belong to a label language $\mathcal{L}$ specified later (in section 3.1.1). Using only simple automata, however, only allows a flat representation of behaviors that is why we call these simple primitive elements *flat* automata. Such a flat representation is not friendly to framework users for describing complex component behavior. We thus need a language that makes it possible to describe complex behavioral entities in a structured way, by means of *scoping* and *hierarchical* composition.
A graphic language is natural when dealing with automata. Our language is very similar to the *Argos* graphical language [36]. It has a well-founded mathematical model and it supports existing verification methods and tools. It offers a graphical notation close to UML statecharts with some restrictions, but with a different semantics based on the synchronous hypothesis. A textual notation is also introduced and detailed later. Programs written in this language operationally describe behavioral entities; we call them *behavioral programs*. We use behavorial programs to represent the state behavior of classes as well as of operations.

3.1.1 Label Language

The label language $\mathcal{L}$ describes the transition labels. These labels correspond to input/output events which determine how the behavioral entity changes its state. In the language, an event is simply represented by a symbol and, thus, it may receive various interpretations. For instance, it may be associated with the code of an operation or with another behavioral program. The syntax of labels is powerful enough to express how the protocol of use of a component works. We denote $\mathcal{E}$ the set of atomic events, which can be:

- symbols expressing event names, variable names, function and method names.
- function calls: Plus(x,y) for instance.
- method calls: Snapshot.memorize() for instance.

From atomic events, labels are defined by the grammar described in table 1.

3.1.2 Definition of the Behavior Description Language

An automaton is a tuple (S, s_0, T) where S is a finite set of states, s_0 is the initial state, T is the transition relation: $T \subseteq Q \times \mathcal{L} \times Q$. As mentioned earlier, each label has two parts: an input part and

label	:=	trigger action_list
trigger	:=	boolean_expression
action_list	:=	$\emptyset$ | action **,** action_list
action	:=	atom
boolean_expression	:=	expression
		boolean_expression **and** boolean_expression
		boolean_expression **or** boolean_expression
		not boolean_expression
expression	:=	atom | atom OP atom
OP	:=	== | <> | <= | >=
atom	:=	$\mathcal{E}$

Table 1: *BDL label grammar:* to express the grammar we use the BNF popular style. Bold **terms** denote lexical entities.

an output part. The input part is called the *trigger* and the transition is fired when the trigger holds. The output part represents a set of actions performed (mainly event emissions). We define the input event set (I_A) of a BDL automaton as follows: let s be a state in S, $I_s = \bigcup\{atom(i) | \exists s \xrightarrow{i/o} s' \in T\}$ and $I_A = \bigcup_{s \in S} I_s$, where $atom(i)$ is the set of atoms in i. This latter function is defined on the syntax of triggers. Similarly, the output event set (O_A) of a BDL automaton is defined by: $O_s = \bigcup\{o | \exists s \xrightarrow{i/o} s' \in T\}$ and $O_A = \bigcup_{s \in S} O_s$.
To describe a complex behavior or a combination of behaviors we propose two main operators: parallel composition and hierarchical refinement of states. When doing refinement together with parallel composition, it is natural to restrict the scope of some signals, hence a third additional operator, the scope operator. The notion of flat automata combined with these three operators is sufficient to induce a hierarchical modular description.
The language operators are:

1. *Parallel composition* is a symmetric operator (noted $\|$ in the textual notation and by a dotted line (aka swim-lane), in the graphical one) which behaves as the synchronous product of its operands[4]. Figure 3 shows the parallel composition of two behavioral programs: the two operands start simultaneously and they react independently. The left one emits `A` when the second occurrence of `a` is received in the global environment, while the right one emits `B` simultaneously with the receipt of the second occurrence of `b`.

2. *Hierarchical composition* corresponds to the possibility for a state in an automaton to be refined by a behavioral (sub) program. If P is a flat automaton with $s_1, ..., s_n$ being some of its states, we can express that state s_i is refined into the behavioral sub-program P_i for $i \in [1, n]$ by the following notation: $P[P_1/s_1, P_2/s_2, ..., P_n/s_n]$ where $/$ reads as "refines". The language enforces a strict containment of sub-programs within their containing state: in

[4] This operation can be considered as "nilpotent": since $P \parallel P$ has the same behavior as P we can consider that they are equivalent ($P \parallel P = P$).

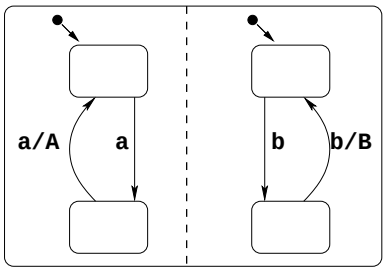


Figure 3: Example of behavioral program with parallel composition the left operand emits A every second occurrence of a. The right one acts similarly

particular, transitions in a sub-program cannot cross the border of the containing state (contrary to StateCharts). Furthermore, this operation supports preemption transitions allowing to express exceptions and normal termination of sub-programs. Exceptions correspond to a preemption fired by an external event, while normal termination corresponds to a preemption fired by an event emitted inside the sub-program. As an example, figure 4 shows an automaton with a refined state. Event b preempts the sub-program refining state A which means that as soon as b occurs, the sub-program terminates and state B is reached (event b is an exception).

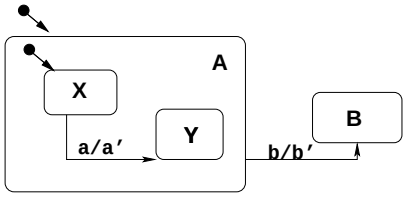


Figure 4: Example of hierarchical composition

3. *Scoping*: This operator, noted $P|_Y$ in the textual form where P is a program and Y a set of *local* events, makes it possible to restrict the scope of some events. Indeed, when refining a state by combining hierarchical and parallel composition, it may be useful to send events from one branch of the parallel composition to the other(s) without these events being globally visible. This operation can be seen as encapsulation: local events that fired a transition must be emitted in their scope; they cannot come from the surrounding environment. Figure 5 shows how the scoping operator works with hierarchical composition. The normal termination of the sub-program that refines state A is achieved by broadcasting event b. As soon as event a is received, b is emitted by the sub-program and raises the preemptive transition of state A. Since b is local, the flat automata associated to this program is reduced to two states and a transition bearing a as label and b no more appear in labels of this resulting automata. On the other hand, figure 6 shows an example of encapsulation for parallel composition, the example describes a

3 bit counter. In this example, b and c are local events used to establish the communication between two parallel composition operands.

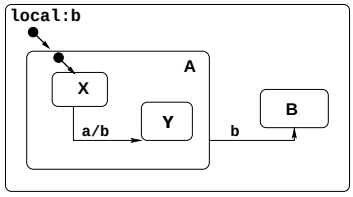


Figure 5: Example of scoping: hierarchical composition encapsulation.

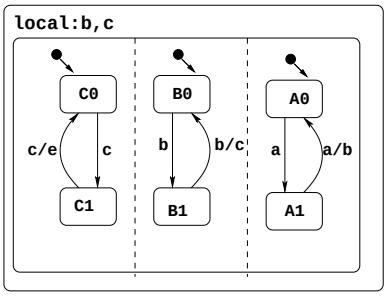


Figure 6: Example of scoping: parallel composition encapsulation

3.2 Mathematical Model of Behavior

Usual mathematical models for synchronous languages are input/output labeled transition systems [36]. Each reaction corresponds to a transition and obeys the synchrony hypothesis. These systems are a special kind of finite deterministic state machines and we shall denote them LFSM for short. Each transition has a *label* representing an elementary execution step of the entity, consisting of a *trigger* (input condition) and an *action* to be executed when the transition is fired. In our case an action corresponds to emitting events, such as calling an operation of some component whereas a trigger corresponds to receiving events such as calling an operation.
A LFSM is a tuple $M = (S, s_0, T, A)$ where S is a finite set of states, $s_0 \in S$ is the initial state, A is the *alphabet* of events from which the set of labels L is built, and T is the transition relation $T \subseteq S \times L \times S$. *We introduce the set I of input events $I \subseteq A$ and the set $O \subseteq A$ of output events (or actions).*

Labels

L, the set of *labels*, has elements of the form i/o, where $i \subseteq I$ is the trigger and $o \subseteq O$ the action or output events set; i has the form $i^+ \cup i^-$ where i^+, the positive (input event) set of a label, consists

of the events tested for their presence in the trigger at a given instant, and i^-, the negative (input event) set, consists of the events tested for their absence at the same instant. Note that I and O need not be disjoint.
A trigger contains the information about all the input events, be they present or absent at a given instant. Obviously, an event cannot be tested for both absence and presence at the same instant. Thus (i^+, i^-) constitutes a partition of I. Moreover, as a consequence of the previous definition of an *instant* in the synchronous model, an event cannot be tested for absence while being emitted in the same instant. Hence, the following well-formedness conditions on labels must hold:

$$\begin{cases} i^+ \cap i^- = \emptyset & \text{(trigger consistency)} \\ i^+ \cup i^- = I & \text{(trigger completeness)} \\ i^- \cap o = \emptyset & \text{(synchrony hypothesis)} \end{cases}$$

Trigger completeness law corresponds to the notion of "completely specified" automaton. It is clear that $i \cap o$ (that is in fact $i^+ \cap o$) can be non empty: indeed in the synchronous paradigm it is possible to test the presence of an event in the same instant it is emitted; it is even the primary way of modeling communication. We call $i^+ - o$ the strict positive trigger part of i; it contains only the input events present in an instant. The well-formedness of labels implies that $(o \cap I) \subseteq i^+$.

Transitions

Each transition has three parts: a source state s, a label l, and a target state s'; $s \xrightarrow{l} s'$ denotes the transition (s, l, s').

A *path* in a LFSM M is a (possibly infinite) sequence of transitions $\pi = s_0 \xrightarrow{i_0/o_0} s_1 \xrightarrow{i_1/o_1} s_2 ...$ such that $\forall i (s_i, i_i/o_i, s_{i+1}) \in T$. The sequence $i_0/o_0, i_1/o_1 ...$ is called the *trace* associated with the path. When such a path exists, the corresponding trigger sequence $i_0, i_1, ...$ is said to be a *valid input sequence* of M.
The LFSMs we consider are deterministic. This property means that there cannot exist two transitions leaving the same state and bearing the same trigger. Formally, if there are two transitions from the same state s such that $s \xrightarrow{i_1/o_1} s_1$ and $s \xrightarrow{i_2/o_2} s_2$, with $s_1 \neq s_2$, then $i_1 \neq i_2$. This requirement for determinism constitutes one of the foundations of the synchronous approach and is mandatory for all models and proofs that follow.

Behavioral Substitutability

The substitutability principle should apply to the dynamic semantics of a behavioral entity–such as either a whole class, or one of its (redefined) operations [23, 43]. If M and M' are LFSMs denoting respectively some behavior in a base class and its redefinition in a derivative, we seek for a relation $M' \preceq M$ stating that "M' extends M in a safe way". To comply with inheritance, this relation must be a preorder.
Following the substitutability principle, we say that M' is a correct extension of M, iff the alphabet of M' (A'_M) is a superset of the alphabet of M (A_M) and every sequence of inputs that is valid for

M is also valid for M' and produces the same outputs (once restricted to the alphabet of M). Thus, the behavior of M' restricted to the alphabet of M is identical to the one of M. Formally,

$$M' \preceq M \Leftrightarrow A_M \subseteq A_{M'} \wedge M \; \mathcal{R}_{Sim} \; (M' \backslash A_M)$$

where $M' \backslash A_M$ is the *restriction* of M' to the alphabet of M and $\mathcal{R}_{Sim}$ is the behavioral simulation relation. Both are defined below.
First, we define the restriction ($l \backslash A$) of a label (l) over an alphabet (A) as follows: let $l = i/o$ and $I(\text{resp } O) \subseteq A$, the input (resp output) event set of A,

$$l \backslash A = \begin{cases} (i \cap I)/(o \cap O) & \text{if } i^+ - o \subseteq I \\ undef & \text{otherwise} \end{cases}$$

Intuitively, this corresponds to consider as undefined all the transitions bearing a strict positive trigger not included in I, and to strip the events not in O from the outputs of the remaining transitions. According to the definition of label, the emitted output events are included in the trigger part of label. The restriction operation only affects the strict positive trigger part.
The restriction of M to the alphabet A (generally with $A \subseteq A_M$) is obtained by restricting all the labels of M to A, then discarding the resulting undefined transitions.
Formally, let $M = (S, s_0, T, A_M)$ be a LFSM,
$M \backslash A = (S, s_0, T \backslash A, A_M \cap A)$ where $T \backslash A$ is defined as follows:

$$s \xrightarrow{l'} s' \in T \backslash A \Leftrightarrow \exists \; s \xrightarrow{l} s' \in T \wedge l' = l \backslash A \neq undef$$

Second, we adopt a behavioral simulation relation similar to Milner's classical simulation [39]. Let M_1 and M_2 be two LFSMs with the same alphabet: $M_1 = (S_{M_1}, s_0^{M_1}, T_{M_1}, A)$ and $M_2 = (S_{M_2}, s_0^{M_2}, T_{M_2}, A)$. A relation $\mathcal{R}_{Sim} \subseteq S_{M_1} \times S_{M_2}$ is called a *simulation* iff $(s_0^{M_1}, s_0^{M_2}) \in \mathcal{R}_{Sim}$ and

$$\forall (s_1, s_2) \in \mathcal{R}_{Sim} :$$
$$s_1 \xrightarrow{l} s_1' \in T_{M_1} \Rightarrow \exists s_2 \xrightarrow{l} s_2' \in T_{M_2} \wedge (s_1', s_2') \in \mathcal{R}_{Sim}$$

Simulation is local, since the relation between two states is based only on their successors. As a result, it can be checked in polynomial time [6], hence, it is widely used as an efficient computable condition for trace-containment. Moreover, the simulation relation can be computed using a symbolic fixed point procedure [25], allowing to tackle large-sized state spaces.
We say that M' *simulates* M iff $M' \preceq M$. Thus, M' simulates M iff there exists a relation binding each state of M to a state of the restriction of M' to the alphabet of M. Any valid sequence of M is also a valid sequence of M' and the output traces are identical, once restricted to A_M. As a consequence, if M' simulates M, M' can be substituted for M, for all purposes of M.
Milner's simulation relation ($\mathcal{R}_{Sim}$) is a preorder and preserves satisfaction of the formulae of a subset of temporal logic, expressive enough for most verification tasks (namely $\forall CTL^*$ [30]). Moreover, this subset has efficient model checking algorithms. Obviously, relation $\preceq$ is also a preorder over LFSMs and any formula that holds for M holds also for M'.
The notion of correct extension can be easily extended to components. We can represent the *protocol of use* of a class C (see section 2.3) by a LFSM $\mathcal{P}(C)$. If C and C' are two classes, $C' \preceq C$ iff (1)

C' derives from C (according to footnote [3], this means "is a subtype of"), (2) the protocol of use of C' simulates the one of C, that is $\mathcal{P}(C') \preceq \mathcal{P}(C)$. As indicated in 2.3, we assume that the protocol of use of a class describes not only the way the other objects may call the class operations, but also the way the operations of the class invoke operations on (other) objects.
With such a model, the description of behavior matches the class hierarchy. Hence, class and operation refinements are compatible and consistent with the static description: checking dynamic behavior may benefit from the static hierarchical organization.

3.3 Behavior Description Language Semantics

The BDL language offers syntactic means to build programs that reflect the behavior of components. Nevertheless, the soundness of this approach implies a clear definition of the relationship between behavioral programs and their mathematical representations as LFSMs (section 3.2). Indeed, BDL flat automata and LFSMs are very close, they correspond to different representations. Flat automata are dedicated to the concrete description of component behavior, while LFSMs are their abstract models. In particular, flat automaton labels refer to a concrete input language while LFSM labels refer to an abstract alphabet.

Label Interpretation

First, we define an *interpretation* function which maps BDL labels to LFSM labels. In particular, this function achieves an *abstraction* of BDL labels, giving them a normal form representation which allows label comparisons. In BDL flat automata, the equality of concrete labels is not easy to express. But in LFSM, since labels refer only to elements of a finite abstract alphabet, label equality is just a syntactic equality. Thus, we consider that two labels in BDL language are equal if and only if their respective abstractions are. In particular, the trigger part of LFSM labels is a set of atomic events (i.e. test of absence or presence of events and elementary conditions), and the transition is fired when the conjunction of all these atomic events is true. Disjunctive BDL transition triggers are not expressed directly; there are interpreted as several labels with only conjunctive trigger parts. The interpretation function is significant only for triggers since the output part of labels is composed of set of atomic events being their own interpretation.
In order to define the interpretation function over triggers, we introduce an operation (denoted $\mathcal{N}$) which converts the trigger parts of BDL labels into disjunctive normal forms. The **or** operators are moved out while the **not** operators are moved inside, in such a way that they concern only atomic boolean expressions[5]. This operation is defined by applying the following equivalence rules:

a <> b ≡ **not** (a==b).

a > b ≡ **not** (a<=b)

a >= b ≡ **not** (a<b)

[5] Atomic boolean expressions are either elements of $\mathcal{E}$ or of the form e OP e', where OP stands for any of the boolean operators: ==, <> , > , < , >= , <=.

not(a **and** b) ≡ (**not** a) **or** (**not** b).

not(a **or** b)≡ (**not** a) **and** (**not** b).

a **and** (b **or** c) ≡ (a **and** b) **or** (a **and** c).

In the sequel, we will denote $\mathcal{N}(i)$ the disjunctive normal form of a trigger i.

Definition of BDL Semantics

Let $\mathcal{P}$ the set of behavioral programs and $\mathcal{M}$ the set of LFSMs. We define a *semantic* function $\mathcal{S} : \mathcal{P} \longrightarrow \mathcal{M}$ that is stable with respect to the previously defined operators (parallel composition, hierarchical composition and scoping). To define this semantic function, we introduce two auxiliary technical operations over labels: completion of a label l with respect to an alphabet A (denoted by $\Phi_A(l)$) and "union" of two labels l_1 and l_2 (denoted by $l_1 \oplus l_2$).

1. $\Phi_A(l)$: let us assume that $l = i/o$ is a well-formed label with respect to a set of events I. $\Phi_A(l) = i'/o$ where ${i'}^+ = i^+ \cup (A \cap o)$ and ${i'}^- = i^- \cup (A - o - {i'}^+)$. By construction, this completion operation results in a well-formed label with respect to $I \cup A$:

$$\begin{aligned} {i'}^+ \cup {i'}^- &= I \cup A \\ {i'}^+ \cap {i'}^- &= \emptyset \\ {i'}^- \cap o &= \emptyset \end{aligned}$$

2. $l_1 \oplus l_2$: let us assume that $l_1 = i_1/o_1$ and $l_2 = i_2/o_2$. l_1 and l_2 are well-formed, let $I_1 = {i_1}^+ \cup {i_1}^-$ and $I_2 = {i_2}^+ \cup {i_2}^-$. Thus, we define $l_1 \oplus l_2 = i/o$ where:

$$\begin{aligned} i^+ &= i_1^+ \cup i_2^+ \cup (I_1 \cap o_2) \cup (I_2 \cap o_1) \\ i^- &= (I_1 \cup I_2) - i^+ \\ o &= o_1 \cup o_2 \end{aligned}$$

   By construction, $l_1 \oplus l_2$ is well-formed.

Now, we present the rules that define the semantic function:

- For a *flat automaton* ($Aut = (S, s_0, T)$):
  $\mathcal{S}(Aut)$ is an LFSM with the same set of states, the same initial state, and the same alphabet as Aut. Its transition relation T_M is the one of Aut where each trigger has been completed with the test of absence of all input events that the corresponding trigger of P does not contain. This satisfies the trigger completeness condition for LFSMs In the behavioral language it is natural to only consider positive triggers, whereas the mathematical model needs to deals with absence of events, hence this completion operation. Formally,
  $\mathcal{S}(Aut) = (S, s_0, T_M, A_M)$, where $A_M = I_{Aut} \cup O_{Aut}$. T_M is defined from T as follows: $\forall (s \xrightarrow{i/o} s') \in T \Rightarrow \{(s \xrightarrow{(i'/o')} s') | i'/o' \in \mathcal{I}_{I_{Aut}}(i/o)\}$. where $\mathcal{I}_{I_{Aut}}(i/o)$ provides a set of

well-formed label in $\mathcal{S}(Aut)$. To define $\mathcal{I}_{I_{Aut}}$, we consider labels in disjunctive normal form, hence $\mathcal{N}(i) = \mathcal{N}(i_1)$**or** $\mathcal{N}(i_2)$. Then, $\mathcal{I}_{I_{Aut}}(i/o) = \{\mathcal{I}_{I_{Aut}}(i_1/o), \mathcal{I}_{I_{Aut}}(i_2/o)\}$. Now, we define $\mathcal{I}_{I_{Aut}}$ for normalized labels without **or** operator:

- if a is an atomic boolean expression in normal form : $\mathcal{I}_{I_{Aut}}(a/o)$ = i_0/o where ${i_0}^+ = \{a\} \cup (I_{Aut} \cap o$ and ${i_0}^- = I_{Aut} - {i_0}^+$.
- if a is an atomic boolean expression in normal form: $\mathcal{I}_{I_{Aut}}(\mathbf{not}\ a/o)$ = i_0/o where ${i_0}^+ = I_{Aut} \cap o$ and ${i_0}^- = I_{Aut} - o$ (a is in I_{Aut}, hence it is included in the negative part of the trigger).
- $\mathcal{I}_{I_{Aut}}(b$ and $c/o)$ = i_0/o where ${i_0}^+ = {i_b}^+ \cup {i_c}^+$ and ${i_0}^- = {i_b}^- \cup {i_c}^- - ({i_b}^+ \cap {i_c}^-) - ({i_c}^+ \cap {i_b}^-)$, where $\mathcal{I}_{I_{Aut}}(b/o)$ = i_b/o and $\mathcal{I}_{I_{Aut}}(c/o)$ = i_c/o

Each label in $\mathcal{I}_{I_{Aut}}(i/o)$ is well-formed by construction: it is obvious for atomic boolean expressions and their negation. For the latter **and** case, the well-formedness of the result is deduced from 1) the respective well-formedness of the sub-terms and 2) the removing of the positive part of a sub-term included in the negative part of the other.

- For *parallel composition* ($(P\|Q)$):
  Let P and Q be two behavioral programs, with $\mathcal{S}(P) = (S_P, s_0^P, T_P, A_P)$ and $\mathcal{S}(Q) = (S_Q, s_0^Q, T_Q, A_Q)$, then $\mathcal{S}(P\|Q) = (S_P \times S_Q, s_0^P \times s_0^Q, T_{P\|Q}, A_P \cup A_Q$ where $T_{P\|Q}$ is defined by the following rules:

  1.
  $$P_1 : \frac{s_1 \xrightarrow{i_P/o_P} s_1' \in T_P, s_2 \xrightarrow{i_Q/o_Q} s_2' \in T_Q}{(s_1, s_2) \xrightarrow{i_P/o_P \oplus i_Q/o_Q} (s_1', s_2') \in T_{P\|Q}}$$

  2.
  $$P_2 : \frac{s_1 \xrightarrow{i_P/o_P} s_1' \in T_P, s_2 \in S_Q}{(s_1, s_2) \xrightarrow{\Phi_{I_Q}(i_P/o_P)} (s_1', s_2) \in T_{P\|Q}}$$

  3.
  $$P_3 : \frac{s_1 \in S_P, s_2 \xrightarrow{i_Q/o_Q} s_2' \in T_Q}{(s_1, s_2) \xrightarrow{\Phi_{I_P}(i_Q/o_Q)} (s_1, s_2') \in T_{P\|Q}}$$

  Rule P_1 characterizes the synchronous hypothesis witch allows the simultaneity of triggers. Here, the trigger of the resulting transition is the union of the respective triggers of each operands. On the other hand, rules P_2 and P_3 correspond to the usual interleaving of the two operands.

  For example, figure 7 shows the LFSM modeling the behavorial program of figure 3.

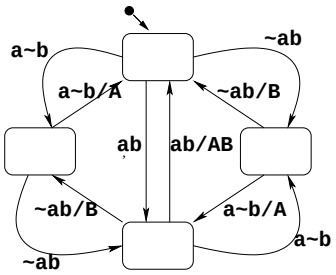


Figure 7: The LFSM associated with the behavioral program of figure 3

- For *hierarchical composition* ($P[Q/s]$):
  Let P and Q be two behavioral programs, with $\mathcal{S}(P) = (S_P, s_0^P, T_P, A_P)$ and $\mathcal{S}(Q) = (S_Q, s_0^Q, T_Q, A_Q)$, then $\mathcal{S}(P[Q/s])$ is $\mathcal{S}(P)$ where state s in P is refined by $\mathcal{S}(Q)$. Informally, s is replaced by $\mathcal{S}(Q)$ and the set of states of $\mathcal{S}(P[Q/s])$ is of the form $S_P \backslash \{s\} \cup \{s.s_i' | s_i' \in S_Q\}$. If $s = s_0^P$, the initial state of $\mathcal{S}(P[Q/s])$ is $s_0^P.s_0^Q$, otherwise it is s_0^P. The set of events is $A_P \cup A_Q$ and the transition relation T' are built from the following rules:

  1. $$R_1 : \frac{s \xrightarrow{i_P/o_P} s_2 \in T_P, s_i' \xrightarrow{i_Q'/o_Q'} s_j' \in T_Q}{s.s_i' \xrightarrow{\Phi_{I_Q}(i_P/o_P)} s_2 \in T'}$$

     Intuitively, a preemption transition can be fired and the ${i_Q'}^+$ does not hold (the internal transition is not fire-able). The preemption of the enclosing state s is done whatever the transitions of Q are and only the external actions are emitted.

  2. $$R_2 : \frac{s \xrightarrow{i_P/o_P} s_2 \in T_P, s_i' \xrightarrow{i_Q'/o_Q'} s_j' \in T_Q}{s.s_i' \xrightarrow{i_P/o_P \oplus i_Q'/o_Q'} s_2 \in T'}$$

     Similarly to the previous rule, a global preemption transition can be fired but ${i_Q'}^+$ holds (the internal transition is fire-able). So, both internal and external actions are simultaneously performed.

  3. $$R_3 : \frac{s_i' \xrightarrow{i_Q'/o_Q'} s_j' \in T_Q, {i_Q'}^+ - o_Q' \not\subseteq I_P}{s.s_i' \xrightarrow{\Phi_{I_P}(i_Q'/o_Q')} s.s_j' \in T'}$$

     No preemption transition is fire-able, hence we keep the internal transition, completed with respect to I_P. Moreover, this rule discards the transitions of Q whose strict positive trigger is included in the input alphabet of P.

4.

$$R_4 : \frac{u \xrightarrow{i_P/o_P} u_1 \in T_P, u \neq s}{u \xrightarrow{\Phi_{I_Q}(i_P/o_P)} u'_1 \in T'}$$

Here, the source state is not the refined state and two cases may occur: if the target state of the transition in P is the refined state ($u_1 = s$), the target state of the resulting transition is the state corresponding to the initial state of Q in the resulting LFSM ($u'_1 = s.s_0^Q$). Otherwise, it is the target state of the initial transition in P ($u'_1 = u_1$).

For instance, figure 8 shows the LFSM model of the program of figure 4.

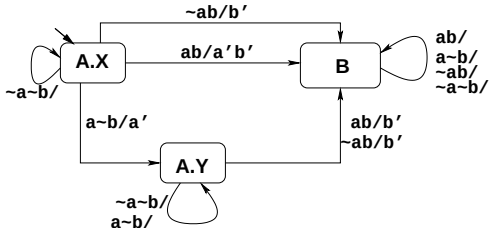


Figure 8: The LFSM associated with the behavioral program of figure 4

After applying the previous rules, the transition relation contains the three following transitions:

1. Preemption b occurs and internal event a does not:
   $A.X \xrightarrow{\sim ab/b'} B$
2. Both preemption b and internal event a occur:
   $A.X \xrightarrow{ab/a'b'} B$
3. Preemption b does not occur and internal event a occurs: $A.X \xrightarrow{a\sim b/a'} A.Y$.

- For *scoping operator* ($P|_Y$):
  $\mathcal{S}(P|_Y)$ is basically $\mathcal{S}(P)$ where some transitions are discarded following a scoping principle and where occurrences of local events are hidden in the labels of the remaining transitions. We define
  $$\mathcal{S}(P|_Y) = (S_P, s_0^P, T', A_P - Y)$$
  where T' is built as follows:
  $$\frac{s \xrightarrow{i_P/o_P} s' \in T_P, i^+ \cap Y \subseteq o}{s \xrightarrow{(i_P/o_P)|_Y} s' \in T'}$$

where $(i_P/o_P)|_Y = (i_P - Y)/(o_P - Y)$. For instance, figure 9 is the model of the program in figure 5, since b is encapsulated, the result when applying the scoping rule is a flat automaton with two states and one transition: A.X $\xrightarrow{a}$ B Similarly, for the program in figure 6, we get the LFSM of figure 10 as model, where b and c no longer appear.

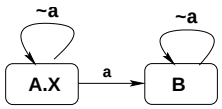


Figure 9: The LFSM associated with the behavioral program of figure 5

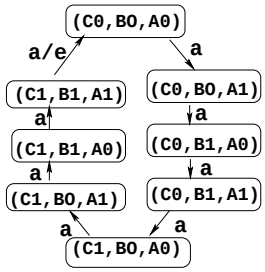


Figure 10: The LFSM associated with the behavioral program of figure 6

Definition: Let P, Q_1 and Q_2 be behavioral programs. We denote I the input event set of $\mathcal{S}(P)$ and O_1 and O_2, the respective output set of $\mathcal{S}(Q_1)$ and $\mathcal{S}(Q_2)$. (P, Q_1) and (P, Q_2) are *local-consistent* if and only if $I \cap O_1 = I \cap O_2$. This definition means that as soon as an output event becomes local in the composition between P and Q_1, it must also be local in the composition between P and Q_2. The following theorem shows that the relation $\preceq$ is a congruence with respect to BDL operators.

Theorem 1 *Let P, Q_1 and Q_2 be local-consistent behavioral programs such that $\mathcal{S}(Q_1) \preceq \mathcal{S}(Q_2)$ and P and Q_1 as well as P and Q_2 have disjoint output sets; the following holds:*

1. $\mathcal{S}(P[Q_1/s]) \preceq \mathcal{S}(P[Q_2/s])$
2. $\mathcal{S}(P \| Q_1) \preceq \mathcal{S}(P \| Q_2)$
3. $\mathcal{S}(Q_1|_Y) \preceq \mathcal{S}(Q_2|_Y)$

This *congruence property* is fundamental to our approach. It gives a modular and incremental way to verify behavioral programs using their natural structure: properties of a whole program can be deduced from properties of its sub-programs. This helps to push back the bounds of state explosion, the major drawback of model checking.

Proof:

Let $\mathcal{S}(Q_1) = \{S_1, s_0^1, T_1, A_1\}$, $\mathcal{S}(Q_2) = \{S_2, s_0^2, T_2, A_2\}$ and $\mathcal{S}(P) = \{S_P, s_0, T, A\}$. Let $I_1(resp.O_1) \subseteq A_1, I_2(resp.O_2) \subseteq A_2, I(resp.O) \subseteq A$ be the input (resp. output) event sets of $\mathcal{S}(Q_1), \mathcal{S}(Q_2)$ and $\mathcal{S}(P)$.
$\mathcal{S}(Q_1) \preceq \mathcal{S}(Q_2)$ means that there is a simulation preorder relation $\mathcal{R}_{Sim}$ between $\mathcal{S}(Q_2)$ and $\mathcal{S}(Q_1)\backslash A_2$ and $A_2 \subseteq A_1$.
Preliminary remark:
$\mathcal{S}(Q_2)\ \mathcal{R}_{Sim}\ \mathcal{S}(Q_1)\backslash A_2$ means that $\forall s_2 \xrightarrow{i_2/o_2} s_2' \in T_2 \Rightarrow \exists s_1 \xrightarrow{i_1/o_1} s_1' \in T_1$ and $\Rightarrow \exists s_2 \xrightarrow{i_2/o_2} s_2' \in T_2$ and $(i_1/o_1)\backslash A_2 = i_2/o_2$. According to the definition of restriction operation we know that $o_1 \cap A_2 = o_2$ and from the local-consistency of (P, Q_1) and (P, Q_2), we deduce $(I \cap o_1) \subseteq (I \cap O_2)$; hence, $(I \cap o_1) = (I \cap o_2)$.

1. *Hierarchical composition:*
   We define the preorder relation ${\mathcal{R}'}_{Sim}$ between $\mathcal{S}(P[Q_2/s])$ and $\mathcal{S}(P[Q_1/s])\backslash A \cup A_2$ by : ${\mathcal{R}'}_{Sim} = \{(q,q)|q \in S_P - \{s\}\} \cup \{(s.s_i, s.s_j)|(s_i, s_j) \in \mathcal{R}_{Sim}\}$. We will prove that it is a simulation preorder. Let $(u, v) \in {\mathcal{R}'}_{Sim}$.

   • First, let us assume that $u \in S_P - \{s\}$. So we are in the case of rule R_4, where $u \neq s$ (the considered state is not the refined state) and thus $v = u$. Let $u \xrightarrow{i_2'/o_2'} u_2'$ be a transition in $\mathcal{S}(P[Q_2/s])$, then by construction of the transition relation in this case, there is a transition $u \xrightarrow{i_P/o_P} u' \in T$ and $(i_2'/o_2') = \Phi_{I_2}(i_P/o_P)$ (applying rule R_4). Similarly, a transition $u \xrightarrow{i_1'/o_1'} u_1'$ exists in $\mathcal{S}(P[Q_1/s]$, i_1'/o_1' being well-formed. We want to check that (1) $u \xrightarrow{i_1'/o_1'\backslash(A \cup A_2)} u_1'$ is a transition in $\mathcal{S}(P[Q_1/s]\backslash(A \cup A_2)$ and (2) $i_1'/o_1'\backslash(A \cup A_2) = i_2'/o_2'$.

   Applying rule R_4 means that $(i_2'/o_2') = \Phi_{I_2}(i_P/o_P)$ and $(i_1'/o_1') = \Phi_{I_1}(i_P/o_P)$. Thus, ${i'}_2^+ = i_P^+ \cup (I_2 \cap o_P)$ and ${i'}_1^+ = i_P^+ \cup (I_1 \cap o_P)$.

   (1)$i_1'/o_1'\backslash(A \cup A_2) \neq undef$: $i_P^+ \subseteq I$, hence $(i_P^+ \cup (I_1 \cap o_P)) - o_P = i_P^+ \subseteq I \subseteq (I \cup I_2$ and the restriction of i_1'/o_1' to $(A \cup A_2)$ is defined.

   (2) We consider $i_1' \cap (I \cup I_2)$. First, we compute the positive part of the intersection. ${i_1'}^+ \cap (I \cup I_2) = (i_P^+ \cup (I_1 \cap o_P)) \cap (I \cup I_2)$. But, $(I_1 \cap o_P) \cap (I \cup I_2)$ is made of (a)the members of $(I_1 \cap o_P)$ belonging to I and they are already in i_P^+ and (b) the members of $(I_1 \cap o_P)$ belonging to I_2 and they are in $(I_2 \cap o_P)$ since $I_2 \subseteq I_1$. Thus, $(i_P^+ \cup (I_1 \cap o_P)) \cap (I \cup I_2) = (i_P^+ \cup (I_2 \cap o_P))$ and ${i_1'}^+ \cap (I \cup I_2) = {i_2'}^+$. Second, we compute the negative part of the intersection: ${i_1'}^- = (I \cup I_1) - {i_1'}^+$, then ${i_1'}^- \cap (I \cup I_2) = ((I \cup I_2) - {i_2'}^+) \cap (I \cup I_2)$. As Just proved, ${i_1'}^+ \cap (I \cup I_2) = {i'}_2^+$. Then $(I \cup I_1) \cap (I \cup I_2) = I \cup I_2$, hence ${i_1'}^- \cap (I \cup I_2) = (I \cup I_2) - {i_2'}^+ = {i_2'}^-$. For the output part ,notice that $o_1' = o_2' = o_P$ by construction, hence $(i_1'/o_1')\backslash(A \cup A_2) = i_2'/o_2'$.

   Now, we have to prove that $(u_2', u_1') \in {\mathcal{R}'}_{Sim}$. Either $u' \neq s$ then $u_1' = u_2' = u'$ and by definition $(u', u') \in {\mathcal{R}'}_{Sim}$, or $u' = s$ and then $u_1' = s.s_0^1$ and $u_2' = s.s_0^2$ and by construction of the preorder relation, $(s.s_0^2, s.s_0^1) \in {\mathcal{R}'}_{Sim}$.

• Second, let us assume that u belongs to the refined state s. Hence, $u = s.s_i^2$ and $v = s.s_i^1$. Let $u \xrightarrow{i'_2/o'_2} u'_2$ be a transition in $\mathcal{S}(P[Q_2/s])$. Then by construction of the transition relation, one of the 3 rules R_1, R_2 or R_3 has been applied. We will consider each of these cases.

(a) If R_1 has been applied, then $\exists s \xrightarrow{i_P/o_P} u' \in T$ and $\exists s_i^2 \xrightarrow{i_2/o_2} s_j^2 \in T_2$. Applying rule R_1 means that $(i'_2/o'_2) = \Phi_{I_2}(i_P/o_P)$. From the simulation preorder from $\mathcal{S}(Q_2)$ to $\mathcal{S}(Q_1)\backslash A_2$ we deduce that $\exists s_i^1 \xrightarrow{i_1/o_1} s_j^1 \in T_1$ and by construction, we apply rule R_1 to build a transition in $\mathcal{S}(P[Q_1/s])$. This transition is of the form: $v \xrightarrow{i'_1/o'_1} v'_1$. We want to prove that: $v \xrightarrow{i'_1/o'_1\backslash(A\cup A_2)} v'_1$ exists and that $i'_1/o'_1\backslash(A\cup A_2) = i'_2/o'_2$. From rule R_1, we know that: $i'_1/o'_1 = \Phi_{I_1}(i_P/o_P)$ and $i'_2/o'_2 = \Phi_{I_2}(i_P/o_P)$ and similarly to the previous case, we prove the result. Moreover, either $u'_2 = v'_1 = u' \neq s$ or $u' = s$, thus $u'_2 = s.s_0^2$ and $v'_1 = s.s_0^1$ since the transition we consider here is preemptive and by definition both (u', u') and $(s.s_0^2, s.s_0^1)$ are in ${\mathcal{R}'}_{Sim}$.

(b) If R_2 has been applied, then $\exists s \xrightarrow{i_P/o_P} u' \in T$ and $\exists s_i^2 \xrightarrow{i_2/o_2} s_j^2 \in T_2$. Applying rule R_2 means that $(i'_2/o'_2) = (i_P/o_P) \oplus (i_2/o_2)$. From the simulation preorder from $\mathcal{S}(Q_2)$ to $\mathcal{S}(Q_1)\backslash A_2$ we deduce that $\exists s_i^1 \xrightarrow{i_1/o_1} s_j^1 \in T_1$ such that $i_1/o_1\backslash A_2 = i_2/o_2$ and by definition, we apply rule R_2 to build the transition $v \xrightarrow{i'_1/o'_1} v'_1$ in $\mathcal{S}(P[Q_1/s])$ where $i'_1/o'_1 = (i_P/o_P) \oplus (i_1/o_1)$.
Notice that by definition of the $\oplus$ operator, ${i'_1}^+ = i_p^+ \cup i_1^+ \cup (I_1 \cap o_P) \cup (I \cap o_1)$ and ${i'_1}^- = (I \cup I_1) - {i'_1}^+$.

First,we must verify that the restriction of the label is defined, i.e $v \xrightarrow{i'_1/o'_1\backslash(A\cup A_2)} v'_1 \neq undef$. From the existence of the simulation preorder $\mathcal{R}_{Sim}$ we deduce that $i_1 \cap I_2 = i_2$ and we know that $i_1^+ - o_1 \subseteq I_2$. As a consequence, $(i_P^+ \cup i_1^+ \cup (I \cap o_1) \cup (I_1 \cap o_P)) - (o_P \cup o_1) \subseteq (I \cup I_2)$ and then $i'_1/o'_1\backslash(A \cup A_2) \neq undef$.
Second, we must prove that $(i'_1/o'_1)\backslash(A \cup A_2) = i'_2/o'_2$. To this aim, we compute $i'_1 \cap (I \cup I_2)$. On one hand, ${i'_1}^+ \cap (I \cup I_2)$ = $(i_p^+ \cup i_1^+ \cup (I \cap o_1) \cup (I_1 \cap o_P)) \cap (I \cup I_2)$.

i. $i_P^+ \cap (I \cup I_2) = i_P^+$;
ii. $i_1^+ \cap (I \cup I_2)$ is composed by i_2^+ (since $i_1^+ \cap I_2 = i_2^+$) and by elements of o_1 that does not belong to I_2 (since $i_1^+ - o_1 \subseteq I_2$). Then, $i_1^+ \cap (I \cup I_2)$ = $i_2^+ \cup X$ where $X \subseteq (I \cap o_1)$. Hence, X is concerned by the following case;
iii. We know that $I \cap o_1$ = $I \cap o_2$ (see the preliminary remark), then $I \cap o_1 \subseteq {i'_2}^+$ and as a consequence, $X \subseteq {i'_2}^+$ too;
iv. $(I_1 \cap o_P) \cap (I \cup I_2) = (I_2 \cap o_P) \cup Z$ (because $I_2 \subseteq I_1$) and Z is composed by elements of o_P belonging to I and by construction they belong to i_P^+;

As a consequence, we have shown that: ${i'_1}^+ \cap (I \cup I_2) = i_P^+ \cup i_2^+ \cup (I \cap o_2) \cup (I_2 \cap o_P) = {i'_2}^+$. On the other hand, ${i'_1}^- \cap (I \cup I_2) = ((I \cup I_1) - {i'_1}^+) \cap (I \cup I_2)$. Since ${i'_1}^+ \cap (I \cup I_2) = {i'_2}^+$ and $I \cup I_2 \subseteq I \cup I_1$, we deduce the result: ${i'_1}^- \cap (I \cup I_2) = {i'_2}^-$.

For the output part, we have to prove that: $(o_P \cup o_1) \cap (O \cup O_2) = o'_2$. $o_P \subseteq O$ then $o_P \cap (O \cup O_2) = o_P$ and $o_1 \cap (O \cup O_2) = (o_1 \cap O_2) \cup (o_1 \cap O) = o_2$ from the existence of preorder relation $\mathcal{R}_{Sim}$ and the hypothesis of output disjointness. Finally, $(s.s_j^2, s.s_j^1)$ belongs to $\mathcal{R}'_{Sim}$ by induction hypothesis and this case is done.

(c) If R_3 has been applied, then $\exists s_i^2 \xrightarrow{i_2/o_2} s_j^2 \in T_2$ and $i_2^+ - o_2 \notin I_P$. Applying rule R_3 means that $(i'_2/o'_2) = \Phi_I(i_2/o_2)$. From the simulation preorder from $\mathcal{S}(Q_2)$ to $\mathcal{S}(Q_1)\backslash A_2$ to $\mathcal{S}(Q_2)$ we deduce that $\exists s_i^1 \xrightarrow{i_1/o_1} s_j^1 \in T_1$ and $i_1^+ - o_1 \subseteq I_2$. Moreover, $i_1 \cap I_2 = i_2$ and $o_1 \cap O_2 = o_2$, then $i_1^+ - o_1 \subseteq i_2 - o_2 \notin I_P$. As a consequence, we also apply rule R_3 to build a transition $v \xrightarrow{i'_1/o'_1} v'_1$ in $\mathcal{S}(P[Q_1/s])$ where $i'_1/o'_1 = \Phi_I(i_1/o_1)$. Similarly to the other cases, we aim at checking that $(i'_1/o'_1)\backslash(A \cup A_2) = i'_2/o'_2$. First of all, we check that $v \xrightarrow{(i'_1/o'_1)\backslash(A\cup A_2)} v'_1$ belongs to the transition relation of $\mathcal{S}(P[Q_1/s])$. By induction, we know that $i_1^+ - o_1 \subseteq I_2$, so $(i_1^+ \cup (I \cap o_1)) - o_1 \subseteq I_2$ and the restriction operation holds. To prove that$(i'_1/o'_1)\backslash(A \cup A_2) = i'_2/o'_2$, first, we verify that $i'_1 \cap (I \cup I_2) = i'_2$. By construction,${i'_1}^+$ and ${i'_1}^-$ are disjoint. We compute ,${i'_1}^+ \cap (I \cup I_2) = (i_1^+ \cup (I \cap o_1)) \cap (I \cup I_2)$. Then, $(i_1^+ \cap (I \cup I_2)) = i_2^+ \cup (i_1^+ \cap I)$. But, $(i_1^+ - o_1) \subseteq I_2$ following restriction definition and $o_1 \cap I = o_2 \cap I$ and so is included in ${i'_2}^+$. Thus, $(I \cap o_1)$ is included in ${i'_2}^+$ and we can deduce the result. Following a similar reasoning as for rule R_2, we deduce than ${i'_1}^- \cap (I \cup I_2) = {i'_2}^-$. Finally, $o'_1 = o_1$ and $o'_2 = o_2$ since the Φ function does not affect the output part of labels. From the existence of relation preorder $\mathcal{R}_{Sim}$, we know that $o_1 \cap O_2 = o_2$. Then, $o_1 \cap (O \cup O_2) = o_2 \cup (o_1 \cap O)$ and since P and Q_1 are output disjoint we can deduce that: $o'_1 \cap (O \cup O_2) = o'_2$ and we get the expected result.

By definition of the semantic function, we have $v'_1 = s.s_j^1$ and $u' = s.s_j^2$ and $(s_j^2, s_j^1 \in \mathcal{R}_{Sim}$. Thus, by definition we set $(s.s_j^2, s.s_j^1 \in \mathcal{R}'_{Sim}$.

To conclude the hierarchical composition case, we note that since $A_2 \subseteq A_1$, we have $A \cup A_2 \subseteq A \cup A_1$.

2. *Parallel composition:*
We define the relation $\mathcal{R}'_{Sim}$ between $\mathcal{S}(P\|Q_1)\backslash A \cup A_2$ and $\mathcal{S}(P\|Q_2)$ by $\mathcal{R}'_{Sim} = \{((s, s_i), (s, s_j))|(s_i, s_j) \in \mathcal{R}_{Sim}\}$. We want to prove that it is a preorder simulation relation.

Let $((s, s_i^2), (s, s_i^1)) \in \mathcal{R}'_{Sim}$. Similarly to hierarchical composition, we will consider the 3 rules applied to build a transition for parallel operator semantics.

(a) We assume that rule P_1 has been applied. Let $(s, s_i^2) \xrightarrow{i'_2/o'_2} (s', s_j^2)$ be a transition in $\mathcal{S}(P\|Q_2)$ corresponding to P_1 application. According to the transition relation construction for this rule, we have $s \xrightarrow{i_P/o_P} s' \in T$, $s_i^2 \xrightarrow{i_2/o_2} s_j^1 \in T_2$ and $i'_2/o'_2 = (i_P/o_P) \oplus (i_2/o_2)$. $\mathcal{S}(Q_1) \preceq \mathcal{S}(Q_2) \Rightarrow \exists\ s_i^1|s_i^1 \xrightarrow{i_1/o_1} s_j^1 \in T_1$ and $i_1/o_1\backslash A_2 = i_2/o_2$. Then,

by construction of the transition relation for parallel composition: $(s, s_i^1) \xrightarrow{i'_1/o'_1} (s', s_j^1)$, where $i'_1/o'_1 = (i_P/o_P) \oplus (i_1/o_1)$ belongs to the transition relation of $\mathcal{S}(P \| Q_1)$. We want to check that $(i'_1/o'_1)\backslash(A \cup A_2) = i'_2/o'_2$. We have the same induction hypothesis than in the previous case of rule R_2, thus we apply a similar reasoning to get the result.

(b) We assume that rule P_2 has been applied. In this case, we assume that we have a transition $(s, s_i^2) \xrightarrow{i'_2/o'_2} (s', s_i^2)$ in $\mathcal{S}(P \| Q_2)$. Then, there is a transition $s \xrightarrow{i_P/o_P} s' \in T$, and $i'_2/o'_2 = \Phi_{I_2}(i_P/o_P)$. Let $s_1 \in T_1$ such that $(s_2, s_1) \in \mathcal{R}_{Sim}$. We also apply rule P_2 to construct a transition: $((s, s_i^1) \xrightarrow{i'_1/o'_1} (s', s_i^1)$ in $\mathcal{S}(P \| Q_1)$ where $i'_1/o'_1 = \Phi_{I_1}(i_P/o_P)$. We must prove that $i'_1/o'_1\backslash(A \cup A_2) = i'_2/o'_2$ in order to prove that there is a corresponding transition to the initial transition we consider in $\mathcal{S}(P \| Q_1)\backslash(A \cup A_2)$. First of all, notice that $o'_1 = o'_2 = o_P$ since the Φ function does not affect the output part of labels. ${i'_1}^+ = i_P^+ \cup (I_1 \cap o_P)$. Then ${i'_1}^+ - o'_1 = {i'_1}^+ - o_p = i_P^+ - o_p \subseteq I \subseteq I \cup I_2$. Thus, $i'_1/o'_1\backslash(A \cup A_2) \neq undef$. From the previous reasoning processus, it is obvious that ${i'_1}^+ = {i'_2}^+$. Then ${i'_1}^- \cap (I \cup I_2) = ((I \cup I_1) - i_P^+) \cap (I \cup I_2)$. Since $I_2 \subseteq I_1$, ${i'_1}^- \cap (I \cup I_2) = (I \cup I_2) - i_P^+ = {i'_2}^-$. Finally, by construction we have $((s', s_i^2), (s', s_i^1)) \in \mathcal{R}'_{Sim}$.

(c) We assume that rule P_3 has been applied. In this case, we assume that we have a transition $(s, s_i^2) \xrightarrow{i'_2/o'_2} (s, s_j^2)$ in $\mathcal{S}(P \| Q_2)$. Then, there is a transition $s_i^2 \xrightarrow{i_2/o_2} s_j^2 \in T_2$, and $i'_2/o'_2 = \Phi_I(i_2/o_2)$. $\mathcal{S}(Q_1) \preceq \mathcal{S}(Q_2) \Rightarrow \exists\ s_i^1 | s_i^1 \xrightarrow{i_1/o_1} s_j^1 \in T_1$ and $i_1/o_1\backslash A_2 = i_2/o_2$. Thus, $i_1^+ - o_1 \subseteq I_2$, $i_1 \cap I_2 = i_2$ and $o_1 \cap O_2 = o_2$ implies that $i_1^+ - o_1 \subseteq i_2^+ - o_2$ According to semantics definition we also apply rule P_3 to construct a transition $(s, s_i^1) \xrightarrow{i'_1/o'_1} (s, s_j^1)$ in $\mathcal{S}(P \| Q_1)$ where $i'_1/o'_1 = \Phi_I(i_1/o_1)$. As usual, we must prove that $i'_1/o'_1\backslash(A \cup A_2) = i'_2/o'_2$ First of all, notice that $o'_1 = o'_2 = o_P$ since the Φ function does not affect the output part of labels. Then, ${i'_1}^+ = i_1^+ \cup (I \cap o_1)$. By induction, we know that $i_1^+ \subseteq I_2$ then ${i'_1}^+ - o'_1 \subseteq (I \cup I_2)$ and $i'_1/o'_1\backslash(A \cup A_2) \neq undef$. This last point implies that $i_1^+ = i_2^+$ and from local-consistency hypothesis, $I \cap o_1 = I \cap o_2$ we have ${i'_1}^+ = {i'_2}^+$. For the negative part of trigger i'_1, we have ${i'_1}^- \cap (I \cup I_2) = ((I \cup I_1) - {i'_1}^+) \cap (I \cup I_2)$. Since $I_2 \subseteq I_1$, ${i'_1}^- \cap (I \cup I_2) = (I \cup I_2) - {i'_2}^+ = {i'_2}^-$. As a result, $i'_1 \cap (I \cup I_2) = i'_2$ and $i'_1/o'_1\backslash(A \cup A_2) = i'_2/o'_2$. Finally, by induction $((s, s_j^2), (s, s_j^1)) \in \mathcal{R}'_{Sim}$.

Moreover, obviously $A \cup A_2 \subseteq A \cup A_1$.

3. *Scoping:*
We define $\mathcal{R}'_{Sim}$ the preorder relation from $\mathcal{S}(Q_1|_Y)\backslash A_2 - Y$ and $\mathcal{S}(Q_2|_Y)$ by $\mathcal{R}'_{Sim} = \{(u, v) | u \in \mathcal{S}(Q_1|_Y), v \in \mathcal{S}(Q_2|_Y), (u, v) \in \mathcal{R}_{Sim}\}$ Let $(u, v) \in \mathcal{R}'_{Sim}$ and let us assume that $u \xrightarrow{i'_2/o'_2} u'$ is in the transition relation of $\mathcal{S}(Q_2|_Y)$. By definition of the scoping operator: $\exists u \xrightarrow{i_2/o_2} u' \in T_2$ where $(i_2^+ \cap Y) \subseteq o_2$ and $(i'_2/o'_2)|_Y = i_2/o_2$.

$\mathcal{S}(Q_1) \preceq \mathcal{S}(Q_2) \Rightarrow \exists v \xrightarrow{i_1/o_1} v' \in T_1$ and $(u', v') \in \mathcal{R}_{Sim}$ (hence, $(i_1/o_1)\backslash A_2 = i_2/o_2$). We must prove: (a) $\exists v \xrightarrow{i'_1/o'_1} v' \in \mathcal{S}(Q_1|_Y)$ and $(i'_1/o'_1)|_Y = i_1/o_1$; (b) the transition $v \xrightarrow{i'_1/o'_1\backslash A_2 - Y} v'$ is defined; (c) $i'_1/o'_1\backslash A_2 - Y = i'_2/o'_2$.

(a) To prove that $v \xrightarrow{i'_1/o'_1} v'$ is a transition belonging to $\in \mathcal{S}(Q_1|_Y)$, we prove that $i_1^+ \cap Y \subseteq o_1$. But, $i_1^+ = (i_1^+ \cap I_2) \cup X$ where X is composed by elements of i_1^+ that does not belong to I_2. On the one hand, $(i_1^+ \cap I_2) = i_2^+$ and by induction we know that $i_2^+ \cap Y \subseteq o_2 \subseteq o_1$. On the other hand, $\mathcal{S}(Q_1) \preceq \mathcal{S}(Q_2) \Rightarrow i_1^+ - o_1 \subseteq I_2$, then $X \subseteq i_1^+ \cap o_1 \subseteq o_1$. As a consequence, $i_1^+ \cap Y \subseteq o_1$ and the considered transition is in the transition relation of $\mathcal{S}(Q_1|_Y)$.

(b) To prove $(i'_1/o'_1)\backslash A_2 - Y \neq undef$ we must check that ${i'_1}^+ - o_1 \subseteq A_2 - Y$. By induction, we know that $i_1^+ - o_1 \subseteq A_2$ then $i_1^+ - Y - o_1 \subseteq A_2 - Y$ and $i_1^+ - Y = {i'_1}^+$ by definition.

(c) We verify $i'_1/o'_1\backslash A_2 - Y = i'_2/o'_2$: First, by induction we know that $i_1^+ \cap I_2 = i_2^+$, hence $(i_1^+ - Y) \cap (I_2 - Y) = (i_2^+ - Y)$ and by construction $(i_1^+ - Y) = {i'_1}^+$ and $(i_2^+ - Y) = {i'_2}^+$. A similar reasoning is applied to prove that ${i'_1}^- \cap (I_2 - Y) = {i'_2}^-$. Second, for the output part: $o_1 \cap O_2 = o_2$ be induction, hence : $(o_1 - Y) \cap (O_2 - Y) = (o_2 - Y)$ and by construction $(o_1 - Y) = o'_1$ and $(o_2 - Y) = o'_2$.

Finally, $A_2 - Y \subseteq A_1 - Y$ since $A_2 \subseteq A_1$.

□

In our specific component framework (BLOCKS) this property is very useful, since one can deal with highly complex global behaviors when they result/provided that they result from composing elementary behaviors which are easy to verify, modify and understand.

4 Modular verification

Model checking of behavioral programs is one of the motivations for our approach. But model checking activity needs a model to describe the system we want to verify, a language to express the properties, and an algorithm to check these properties on the models. In our approach, we adopt the usual framework for model checking: finite state machines as models and *temporal logic* as language to express properties to be proved. In such a framework, well known and efficient algorithms exist. Temporal logic is a formalism for describing sequences of transitions between states in a synchronous model. In temporal logics time is not explicitly mentioned; but a formula may specify that a particular event will *eventually* occur or that an error event will *never* happen. For instance, properties like *eventually* or *never* are specified by *temporal operators*.These operators can be combined with boolean connectives and nested. The logic is interpreted over "Kripke structures" (see 4.1.1) in order to express model checking algorithms and satisfaction of a state formula is defined in a natural inductive way (see [30] for complete definitions). An LFSM can be mapped to a Kripke structure, which is also a state machine.

Moreover, to avoid the state explosion problem involved in model checking techniques, we focus on a modular approach in the verification process. Such an approach consists to use the natural decomposition of a system. The goal is to decompose the properties of a system into properties of its components, so that if each component verifies its properties, so does the global system. This approach is applied in the "assume-guarantee" paradigm defined by Pnueli. To apply such a paradigm, we have first to check if a component M satisfies a property ϕ, if it is part of a system that satisfies an assumption A. Then, to complete the proof, it must be shown that the remaining components of the system verify A. A way to apply the assume-guarantee paradigm is to define a preorder relation on the component models which preserves the satisfaction of logic temporal formulas. In [30], Clarke and al use Kripke structures and define a preorder ensuring the satisfaction of temporal logic formulas. We totally rely on their work. We will associate a Kripke structure to each LFSM and we will introduce a simulation equivalence between LFSMs such that if two LFSM M and M' are simulation equivalent, their corresponding Kripke structures belong to the preorder relation defined over Kripke structures. In this section, we first describe the approach of Clarke and al and then we will show the relationship between their Kripke structures and our LFSMs.

4.1 Verification context

In this section, we formally introduce the temporal logic we consider and its Kripke structure model.

4.1.1 Kripke Structure

Kripke structures are verification models where model-checking algorithms are defined.
A *Kripke structure* K is a tuple: $K = (S, IS, A, R, L)$ where:

1. S is a finite set of states
2. $IS \subseteq S$ is the set of initial states
3. A is a finite set of atomic propositions
4. $R \subseteq S \times S$ is a transition relation that must be total: for every state $s \in S$, there is a state s' such that $R(s, s')$
5. $L : S \longrightarrow 2^A$ is a labeling function that labels each state by the set of atomic propositions true in that state.

Let K be a Kripke structure, a *path* in K is an infinite sequence of states: $\pi = s_0, s_1, s_2, ...$ such that $\forall i \in \mathcal{N}, R(s_i, s_{i+1})$. Moreover, the notation π^n will denote the suffix of π beginning at s_n.
We say that a Kripke structure K satisfies a state formula ψ ($K \models \psi$) if property ψ is true for the initial state of K. This definition is extended to LFSMs: $M \models \psi$ iff $K \models \psi$ K being the Kripke structure to which M is mapped.

LFSMs to Kripke structures

Let $M =< S, s_0, T, A >$ be a LFSM, a *terminal* state is a state $s \in S$ such that $\nexists s \xrightarrow{i/o} s' \in T$. The Kripke structure $\mathcal{K}(M)$ associated with M is defined as follows: $\mathcal{K}(M) =< S^K, S_0^K, A^K, R^K, L^K >$ where

1. $S^K \subseteq S \times 2^A : S^K = \{(s,v) | \exists (s \xrightarrow{i/o} s' \in T \text{ and } i^+ \cup o = v\} \cup \{(s,\emptyset) | s \in S\}$.
2. $IS^K = (\{s_0\} \times 2^A) \cap S^K$
3. $A^K = A$
4. $L^K(s,v) = v$
5. $R^K((s,v),(s',v'))$ iff $\exists s \xrightarrow{i/o} s'$ and $v = i^+ \cup o$ and $(s',v') \in S^K$ and $R^K((s,\emptyset),(s,\emptyset))$ for $s \in S^K$.

4.1.2 Temporal logic

The logic we consider ($\forall CTL^*$) is a formal language where assertions related to behavior are easily expressed. It is based on first-order logic but, in order to be efficient when deciding whether a formula is true, the existential path quantifier has been eliminated. It offers temporal operators that make it possible to express properties holding for a given state, for the next state (operator **X**), eventually for a future state (**F**), for all future states (**G**), or that a property remains true until some condition (**U**). We can also express that a property holds for all the paths starting in a given state ($\forall$). Formally, the logic $\forall CTL^*$ we consider is the set of state formulas defined as follows:

- The constants *true* and *false* are *state* formulas.
- If ψ and ϕ are *state* formulas, then $\psi \vee \phi$ and $\psi \wedge \phi$ are *state* formulas.
- If ϕ is a *path* formula then $\forall(\phi)$ is a *state* formula.
- If ϕ is a *state* formula then ϕ is also a *path* formula.
- If ψ and ϕ are *path* formulas, then $\psi \vee \phi$ and $\psi \wedge \phi$ are *path* formulas.
- If ψ and ϕ are *path* formulas. then so are:
  - $\mathbf{X}\phi$
  - ϕ **U** ψ

Two abbreviations are used: $\mathbf{F}\phi$ and $\mathbf{G}\phi$ (ϕ is a path formula) respectively denote ($true$ **U** ϕ).

Satisfaction of formulas

Now, we introduce the notion of "satisfaction of a formula". We consider that the atomic propositions are in A. The satisfaction of a state formula (ϕ) by a state s (denoted $s \models \phi$) or of a path formula ψ by a path π (denoted $\pi \models \psi$ is inductively defined as follows:

- $s \models true$, $s \not\models false$, $s \models p$ iff $p \in L(s)$ and $s \not\models p$ iff $p \notin L(s)$ [6].
- $s \models \psi \vee \phi$ iff $s \models \psi$ or $s \models \phi$, $s \models \psi \wedge \phi$ iff $s \models \psi$ and $s \models \phi$.
- $s \models \forall(\phi)$ iff for every path π starting at s, $\pi \models \phi$.
- $\pi \models \phi$ where ϕ is a state formula, iff the first state of π satisfies ϕ.
- $\pi \models \psi \vee \phi$ iff $\pi \models \psi$ or $\pi \models \phi$, $\pi \models \psi \wedge \phi$ iff $\pi \models \psi$ and $\pi \models \phi$.
- If ψ and ϕ are *path* formulas:
  - $\pi \models \mathbf{X}\phi$ iff $\pi^1 \models \phi$.
  - $\pi \models \phi \mathbf{U} \psi$ iff $\exists n \in \mathcal{N}$ such that $\pi^n \models \psi$ and $\forall i \leq n, \pi^i \models \phi$.

4.2 Simulation and composition of Kripke structures

In this section, we define the simulation preorder relation over Kripke structures and also a composition operation allowing the decomposition of verification process.

K-simulation in Kripke structure model

Let $K =< S, IS, A, R, L >$ and $K' =< S', IS', A', R', L' >$ be two Kripke structures with $A' \subseteq A$. A relation $H \subseteq S \times S'$ is a K-simulation from K to K' iff $\forall s, s', H(s, s') \Rightarrow$

1. $L(s) \cap A' = L'(s')$
2. For every state s_1 such that $R(s, s_1)$, there is a state s'_1 such that $R(s', s'_1)$ and $H(s_1, s'_1)$.

Moreover, we say that $K \preceq_K K'$ iff there exists a K-simulation relation H such that for every $s_0 \in IS$, there is $s'_0 \in IS'$ and $H(s_0, s'_0)$.

Composition of Kripke structures

Let K and K' be two Kripke structures defined as previously. The composition of these two Kripke structures (denoted $K \| _K K'$) is a Kripke structure $< S_\|, S_{\|_0}, A_\|, R_\|, L_\| >$ built as follows:

- $S_\| = \{(s, s') | L(s) \cap A' = L(s') \cap A\}$
- $IS_\| = (IS \times IS') \cap S_\|$

[6] $L(s)$ is the set of atomic propositions associated with s in the Kripke structure to which s belongs.

- $A_{\|} = A \cup A'$
- $L_{\|}(s, s') = L(s) \cup L'(s')$
- $R_{\|}((s, s'), (t, t'))$ iff $R(s, t)$ and $R'(s', t')$

4.3 Temporal logic formula Verification

4.3.1 Preservation of temporal logic formulas

In [30], two main results are proved. The first one shows that the relation $\preceq_K$ is a preorder relation, and the second one states that this preorder relation preserves the satisfaction of logic temporal formulas.
Let K and K' be two Kripke structures, the following properties hold:

1. $\preceq_K$ is a preorder
2. $K \|_K K' \preceq_K K$
3. for all K'', if $K \preceq_K K'$ then $K \|_K K'' \preceq_K K' \|_K K''$
4. $K \preceq_K K \|_K K$

Then, the second result of Clark and al approach is the following:
Let K and K' be two Kripke structures such that $K \preceq_K K'$. Then for every $\forall CTL*$ formula ϕ (with atomic propositions in A'), $K' \models \phi$ implies $K \models \phi$.
Now, relying on this last property, we will show that we can get a modular approach to verification of $\forall CTL*$ formulas. The $\preceq$ preorder defined over LFSMs enforces the behavioral substitutability required to get a safe subtyping. Nevertheless it is not sufficient to ensure temporal logic formulas preservation. Thus, to solve the problem, we introduce a stronger notion of preorder $\preceq_E$ over LFSMs: $M' \preceq_E M$ if and only if M' restricted to the alphabet of M is equivalent to M. Formally:

$$M' \preceq_E M \Leftrightarrow A_M \subseteq A_{M'} \wedge M \, \mathcal{E}_{Sim} \, (M' \backslash A_M)$$

where $\mathcal{E}_{Sim}$ is a *simulation equivalence*. For two LFSMs M_1 and M_2, $M_1 \preceq_E M_2$ if and only if, there is a pair (R^0_{Sim}, R^1_{Sim}) of simulation relations such that $M_1 R^0_{Sim} M_2$ and $M_2 R^1_{Sim} M_1$.

Proposition 1 *Let M_1 and M_2 be two LFSMs, $M_1 \preceq_E M_2 \Rightarrow \mathcal{K}(M_1 \backslash A_{M_2}) \preceq_K \mathcal{K}(M_2)$.*

Proof

Let $M_1 =< S_1, s^1_0, T_1, A_1 >$ and $M_2 =< S_2, s^2_0, T_2, A_2 >$. We will denote $\mathcal{K}(M_1 \backslash A_2) = < S^K_1, IS_1, A_2, R_1, L_1 >$ and $\mathcal{K}(M_2) = < S^K_2, IS_2, A_2, R_2, L_2 >$. By construction, we recall that $IS_1 = \{s^1_0\} \times 2^{A_2} \cap S^K_1$ and $IS_2 = \{s^2_0\} \times 2^{A_2} \cap S^K_2$ and we have to prove that there exists a relation H such that for every (s^1_0, v_1) in IS_1, there is a state (s^2_0, v_2) and $H((s^1_0, v_1), (s^2_0, v_2)$.
$M_1 \preceq_E M_2$ means that there exists two simulation relations $\mathcal{R}^0_{Sim}$ between M_2 and $M_1 \backslash A_2$ and $\mathcal{R}^1_{Sim}$ between $M_1 \backslash A_2$ and M_2.

We set $H = \{((s_1, v), (s_2, v))|(s_1, s_2) \in \mathcal{R}^1_{Sim}\}$. We must prove that H is a K-simulation between $\mathcal{K}(M_1 \backslash A_2)$ and $\mathcal{K}(M_2)$, i.e we prove the two following items:

1. $\forall (s_1, v), (s_2, v)$ such that $H((s_1, v), (s_2, v))$, if $R_1((s_1, v), (s'_1, v'))$ then there is $(s'_2, v') \in S^K_2$ such that $R_2((s_2, v), (s'_2, v'))$ and $H((s'_1, v'), (s'_2, v'))$
2. $\forall (s^1_0, v) \in IS^K_1$, there is a state $(s^2_0, v) \in IS^K_2$ such that $H((s^1_o, v), (s^2_o, v))$.

(1) $H((s_1, v), (s_2, v))$ implies that (s_1, s_2) belongs to $\mathcal{R}^1_{Sim}$. Let us assume that $R_1((s_1, v), (s'_1, v'))$, thus by definition of $\mathcal{K}$, there is a transition $s_1 \xrightarrow{i/o} s'_1 \in T_1 \backslash A_2$ such that $i^+ \cup o = v$. Since, $\mathcal{R}^1_{Sim}(s_1, s_2)$ holds, there also exists a transition $s_2 \xrightarrow{i/o} s'_2 \in T_2$ and $\mathcal{R}^1_{Sim}(s'_1, s'_2)$. We must prove that $H((s'_1, v'), (s'_2, v'))$. Notice that (s'_1, v') being a state in S^K_1 means that there is a transition $s'_1 \xrightarrow{i'/o'} u'_1$ in $T_1 \backslash A_2$ where $i'^+ \cup o' = v'$. Since, $\mathcal{R}^1_{Sim}(s'_1, s'_2)$ holds there is a transition $s'_2 \xrightarrow{i'/o'} u'_2$ in T_2 and then from the definition of $\mathcal{K}$ we deduce that $H((s'_1, v'), (s'_2, v'))$ holds.
(2) By definition $(s^1_0, s^2_0) \in \mathcal{R}^1_{Sim}$. On the other hand, $(s^1_0, v) \in IS^K_1$ means that there is a transition $s^1_0 \xrightarrow{i_1/o_1} s_1 \in T_1$ such that $(i_1/o_1) \backslash A_2$ is defined and $s^1_0 \xrightarrow{i/o} s_1$ where $(i_1/o_1) \backslash A_2 = i/o$ is a transition in $T_1 \backslash A_2$. According to the definition of simulation, there is a transition $s^2_0 \xrightarrow{i/o} s_2 \in T_2$ and $(s_1, s_2) \in \mathcal{R}^1_{Sim}$ and by definition of H , $H((s^1_0, v), (s^2_0, v))$ holds.
□

This result shows that if two LFSMs are such that the restriction of one to the alphabet of the other is simulation equivalent to the latter, then temporal logic properties are preserved and this is a practical means to ensure safe derivation.

4.3.2 Compositionality and verification of temporal logic properties

In this section, we aim at proving that temporal logic properties are preserved by the operators of the BDL language. First, we prove that these operators preserve the $\preceq_E$ preorder of LFSMs models.
Definition: Let P_1 and P_2 be behavioral programs. We denote I_1 and I_2 the respective input event set of $\mathcal{S}(P_1)$ and $\mathcal{S}(P_2)$.
We say that P_1 and P_2 are *input consistent* if and only if for each triggers i_1 and i_2 respectively in the transition relation of $\mathcal{S}(P_1)$ and $\mathcal{S}(P_2)$, the following property holds: $(i^+_1 - i^+_2) \cap I_2 = \emptyset$ and $(i^+_2 - i^+_1) \cap I_1 = \emptyset$.
Similarly, P_1 and P_2 are *output consistent* if and only if for each output sets o_1 and o_2 respectively in the transition relation of $\mathcal{S}(P_1)$ and $\mathcal{S}(P_2)$, the following property holds: $(o_1 - o_2) \cap I_2 = \emptyset$ and $(o_2 - o^+_1) \cap I_1 = \emptyset$.
These definitions are useful to prove proposition 2. They mean that P_1 and P_2 cannot react differently on the same triggers in different instants. These definitions ensure that the restriction operation behaves as a projection. In fact, when applying label operations (either $\oplus$ or Φ operations), elements not belonging to the labels involved in the operation are not added.

Proposition 2 *Let P and Q be two behavioral programs with disjoint output sets.*

1. $\mathcal{S}(P[Q/s]) \preceq_E \mathcal{S}(P)$

2. *if P and Q are both input and output consistent then $\mathcal{S}(P\|Q) \preceq_E \mathcal{S}(P)$*

Proof

Hierarchical composition
To prove the proposition for hierarchical composition, we build a pair of simulation relations $(\mathcal{R}_{Sim}, \mathcal{R}^1_{Sim})$ such that $\mathcal{S}(P)\ \mathcal{R}_{Sim}\ (\mathcal{S}(P[Q/s]))\backslash A_P$ and $(\mathcal{S}(P[Q/s]))\backslash A_P\ \mathcal{R}^1_{Sim}\ \mathcal{S}(P)$
Let $\mathcal{S}(P) = \{S_P, s_0^P, T_P, A_P\}$ and $\mathcal{S}(Q) = \{S_Q, s_0^Q, T_Q, A_Q\}$ be the respective models of P and Q.

1. $\mathcal{R}_{Sim}$
   We set $\mathcal{R}_{Sim} = \{(u,u)|u \in S_P - \{s\}\} \cup \{(s, s.s_0^Q)\}$ and we show that it is a simulation preorder relation. Thus, let $u \xrightarrow{i_P/o_P} u'$ be a transition in T_P, we must prove that $\exists u \xrightarrow{i_P/o_P} v$ in $(\mathcal{S}(P[Q/s]))\backslash A_P$ such that $(u', v) \in \mathcal{R}_{Sim}$. First, we consider that $u \neq s$. According to the semantics definition, rule R_4 is applied and a transition: $u \xrightarrow{i'/o'} v$ is build in $\mathcal{S}(P[Q/s]$ and $i'/o' = \Phi_{I_Q}(i_P/o_P)$. Function Φ does not affect output sets and thus $o' = o_P$. Thus, $i'^+ = i_P^+ \cup (o_P \cap I_Q)$ and $i'^+ - o_P = i_P^+$, then $i'^+ - o_P \subseteq I_P$ and $i'/o'\backslash A_P \neq undef$. Now , we show that $(i'/o')\backslash A_P = i_P/o_P$. $i'^+ \cap I_P = (i_P^+ \cup (I_Q \cap o_P)) \cap I_P = i_P^+$. For the negative part of i', $i'^- \cap I_P = (i_P^- \cup (I_Q - o_P - i'^+) \cap I_P$ and then $i'^- \cap I_P = i_P^-$ because the elements of $(I_Q - o_P - i'^+) \cap I_P$ are already in i_P^- since i_P constitute a partition of I_P. As a result, we deduce that $(i'/o')\backslash A_P = i_P/o_P$ and then the transition $u \xrightarrow{(i'/o')\backslash A_P} u'$ is a transition of $\mathcal{S}(P[Q/s])\backslash A_P$. On the other hand, either $u' \neq s$ and by construction $v = u'$ or if $u' = s$ then $v = s.s_0^Q$ and by construction (u', u') and $(s, s.s_0^Q)$ belongs to $\mathcal{R}_{Sim}$. Second, if $u = s$, then the transition we consider is a preemptive one and according to rule R_1, we build a transition $s.s_0^Q \xrightarrow{i'/o'} u'$ in $(\mathcal{S}(P[Q/s]))$ and $i'/o' = \Phi_{I_Q}(i_P/o_P)$. Thus, with a similar reasoning as previously, we show that $(i'/o')\backslash A_P = i_P/o_P$ and $(u', u') \in \mathcal{R}_{Sim}$ by definition of the relation. Then, the relation $\mathcal{R}_{Sim}$ is a simulation preorder and to conclude this case we just remark that $A_P \cup A_Q \subseteq A_P$ and thus the alphabet of $\mathcal{S}(P[Q/s])$ is included in the one of $\mathcal{S}(P)$.

2. $\mathcal{R}^1_{Sim}$
   We set $\mathcal{R}^1_{Sim} = \{(u,u)|u \in S_P - \{s\}\} \cup \{(s.s_0^Q, s)\}$ and we prove that it is a simulation relation between $(\mathcal{S}(P[Q/s]))\backslash A_P$ and $(\mathcal{S}(P)$. First, let $u \xrightarrow{i/o} v$ be a transition in $(\mathcal{S}(P[Q/s]))\backslash A_P$ where $u \neq s$. There is a transition $u \xrightarrow{i'/o'} v$ in $(\mathcal{S}(P[Q/s]))$ and $i/o = (i'/o')\backslash A_P$. We want to prove that $u \xrightarrow{i/o} v$ is also a transition in $(\mathcal{S}(P)$. In this case ($u \neq s$), we applied rule R_4 to build the transition in $(\mathcal{S}(P[Q/s]))$ and according to this rule, there is a transition $u \xrightarrow{i_P/o_P} v'$ in T_P such that $i'/o' = \Phi_{I_Q}(i_P/o_P)$. We compute $i/o = i'/o'\backslash A_P$. $i/o = \Phi_{I_Q}(i_P/o_P)\backslash A_P$ then $i^+ = i_P^+ \cup (I_Q \cap o_P)$. But $o_P \subseteq i_P^+$ by well-formedness of labels and so $i^+ = i_P^+$. Then it is obvious that $i^- = i_P^-$ and $o = o_P$. As a

consequence, the transition $u \xrightarrow{i/o} v$ is in T_P. Applying rule R_4 means that either $v = v' \neq s$ or $v' = s$ and $v = s.s_0^Q$. In both cases, (v, v) and $(s.s_0^Q.s)$ are in $\mathcal{R}^1_{Sim}$.

Second, we prove that when rule R_3 is applied to build a transition in $(\mathcal{S}(P[Q/s]))$, this transition does not belong to the restriction of the model to A_P. Assume that there is a transition $s.s'_i \xrightarrow{i'/o'} s.s'_j$ in $(\mathcal{S}(P[Q/s]))$. Then , there is a transition $s'_i \xrightarrow{i_Q/o_Q} s'_j$ in T_Q and $i'/o' = \Phi_{I_P}(i_Q/o_Q)$. Thus $i'^+ = i_Q^+ \cup (I_P \cap o_Q)$. Then $i'^+ - o' = i_Q^+ \cup (I_P \cap o_Q) - o_Q = i_Q^+ - o_Q$. But, apply rule R_3 means that $i_Q^+ - o_Q \notin I_P$ hence $i'^+ - o' \notin I_P$ and the transition is $undef$ and does not belong to $(\mathcal{S}(P[Q/s]))\backslash A_P$. Then, if we consider a transition in $(\mathcal{S}(P[Q/s]))\backslash A_P$, it is issued from $s.s_0^Q$ since the internal transition of $(\mathcal{S}(P[Q/s]))$ are cut by the restriction operation. Hence, we consider a transition $s.s_0^Q \xrightarrow{i/o} v$ in $(\mathcal{S}(P[Q/s]))\backslash A_P$ and it is a preemptive one i.e either rule R_1 or rule R_2 has been applied. Assume that we applied rule R_1. There are transitions $s \xrightarrow{i_P/o_P} s' \in T_P$ and $s_0^Q \xrightarrow{i_Q/o_Q} s'_Q \in T_Q$ and $i/o = \Phi_{I_Q}(i_P/o_P)\backslash A_P$. With a similar reasoning to case of rule R_4, we prove that $i/o = i_P/o_P$. Now, assume that we have applied rule R_2.There are transitions $s \xrightarrow{i_P/o_P} s' \in T_P$ and $s_0^Q \xrightarrow{i_Q/o_Q} s'_Q \in T_Q$ and $i/o = (i_P/o_P \oplus i_Q/o_Q)\backslash A_P$. This transition is defined, so $i^+ - o \subseteq I_P$. But, $i^+ = i_P^+ \cup i_Q^+ \cup (o_Q \cap I_P) \cup (o_P \cap I_Q)$ and $o = o_P \cup o_Q$. Hence, $i^+ - o = i_P^+ \cup i_Q^+ \cup (o_Q \cap I_P) \cup (o_P \cap I_Q) - (o_P \cup o_Q) = i_P^+ \cup i_Q^+ = i_P^+$ since P and Q are output disjoint. i is a partition of $I_P \cup I_Q$, i_Q is a partition of I_Q and i_P a partition of I_P, then $i^- \cap I_P = i_P^-$. For the output part, we have $(o_P \cup o_Q) \cap O_P = o_P$ since P and Q are output disjoint too. As a consequence, $i/o = i_P/o_P$ and the considered transition belongs to $(\mathcal{S}(P)$.

Parallel composition

To prove the proposition for parallel composition, we build a pair of simulation relations $(\mathcal{R}_{Sim}, \mathcal{R}^1_{Sim})$ such that $\mathcal{S}(P)\ \mathcal{R}_{Sim}\ (\mathcal{S}(P\|Q))\backslash A_P$ and $(\mathcal{S}(P\|Q)\backslash A_P\ \mathcal{R}^1_{Sim}\ \mathcal{S}(P)$.
Let $\mathcal{S}(P) = \{S_P, s_0^P, T_P, A_P\}$ and $\mathcal{S}(Q) = \{S_Q, s_0^Q, T_Q, A_Q\}$ be the respective models of P and Q.

1. $\mathcal{R}_{Sim}$
   We set $\mathcal{R}_{Sim} = \{(s_P, (s_P, s_Q))|(s_P, s_Q) \in \mathcal{S}(P\|Q)\backslash A_P\}$. We show that $\mathcal{R}_{Sim}$ is a simulation preorder between $\mathcal{S}(P)$ and $(\mathcal{S}(P\|Q)\backslash A_P$. Thus, let $u \xrightarrow{i_P/o_P} u'$ be a transition in T_P, we must prove that $\exists u \xrightarrow{i_P/o_P} v$ in $(\mathcal{S}(P[Q/s]))\backslash A_P$ such that $(u', v) \in \mathcal{R}_{Sim}$.

   To build the transition relation of $(\mathcal{S}(P\|Q)$, we apply rule P_2 and we get a transition $(u, v) \xrightarrow{i'/o'} (u', v)$ where $i'/o' = \Phi_{I_Q}(i_P/o_P)$ for some $v \in T_Q$. We compute i'/o'. By definition of Φ, $o' = o_P$ and $i' = i_P^+ \cup (I_Q \cap o_P)$. Thus, $i'^+ - o_P \subseteq I_P$ and $i'/o'\backslash A_P \neq undef$. Moreover, $i'^+ \cap I_P = i_P^+ \cup (I_Q \cap o_P) = i_P^+$ since i_P/o_P is well-formed and $o_P \cap I_P \subseteq i_P^+$. For the negative part of i', $i'^- \cap I_P = (i_P^- \cup (I_Q - o_P - i'^+) \cap I_P$ and then $i'^- \cap I_P = i_P^-$ because the elements of $(I_Q - o_P - i'^+) \cap I_P$ are yet in i_P^- since i_P constitute a partition of I_P. As

a result, we deduce that $(i'/o')\backslash A_P = i_P/o_P$ and then the transition $u \xrightarrow{(i'/o')\backslash A_P} u'$ is a transition of $\mathcal{S}(P[Q/s])\backslash A_P$. Finally, $(u', v) \in \mathcal{R}_{Sim}$ by construction.

2. $\mathcal{R}^1_{Sim}$
We set $\mathcal{R}^1_{Sim} = \{((s_P, s_Q), s_P) | (s_P, s_Q) \in \mathcal{S}(P\|Q)\backslash A_P\}$. We show that $\mathcal{R}^1_{Sim}$ is a simulation preorder between $(\mathcal{S}(P\|Q)\backslash A_P$ and $\mathcal{S}(P)$. Assume that there is a transition: $(s_P, s_Q) \xrightarrow{i'_P/o'_P} (s'_P, s'_Q)$ in the transition relation of $(\mathcal{S}(P\|Q)\backslash A_P$. Then, there is a transition: $(s_P, s_Q) \xrightarrow{i'/o'} (s'_P, s'_Q)$ in the transition relation of $(\mathcal{S}(P\|Q)$ and $i'_P/o'_P = i'/o'\backslash A_P$. We want to prove that $s_P \xrightarrow{i'_P/o'_P} s'_P \in T_P$. To build the transition in $(\mathcal{S}(P\|Q)$, we applied either of P_1, P_2 or P_3 rules.

   (a) Let us assume that we applied P_1 to build the transition: $(s_P, s_Q) \xrightarrow{i'/o'} (s'_P, s'_Q)$. Then, there are two transitions $s_P \xrightarrow{i_P/o_P} s'_P \in T_P$ and $s_Q \xrightarrow{i_Q/o_Q} s'_Q \in T_Q$ such that $i'/o' = (i_P/o_P) \oplus (i_Q/o_Q)$. By hypothesis we know that $i'^+ - o' \subseteq I_P$, thus $i^+_P \cup i^+_Q \cup (I_P \cap o_Q) \cup (I_Q \cap o_P) - (o_P \cup o_Q) \subseteq I_P$. But, $(i^+_Q - i^+_P) \cap I_P = \emptyset$ thus $(i^+_Q \cap I_P) \subseteq i^+_P$. As a consequence, $i'^+_P = i' \cap I_P = i^+_P \cup (I_P \cap o_Q)$. We consider $o_Q = (o_Q \cap o_P) \cup (o_Q - o_P)$. On one hand, elements of $(o_Q \cap o_P) \cap I_P$ are already in i^+_P by construction. On the other hand, $(o_Q - o_P) \cap I_P = \emptyset$ since P and Q are output consistent. As a result, $i'^+_P = i^+_P$, then since labels are well-formed, $i'^-_P = i^-_P$ and thus $i'_P = i_P$ and $o'_P = o_P$. To complete this case, we can remark that $((s'_P, s_Q), s'_P) \in \mathcal{R}^1_{Sim}$.

   (b) Let us assume that we applied P_2 to build the transition: $(s_P, s_Q) \xrightarrow{i'/o'} (s'_P, s_Q)$.Then, there is a transition $s_P \xrightarrow{i_P/o_P} s'_P \in T_P$ such that $i'/o' = \Phi_{I_Q}(i_P/o_P)$. To check that $s_P \xrightarrow{i'_P/o'_P} s'_P \in T_P$, we must prove that $(i'_P/o'_P) = i_P/o_P$. $i'^+_P = i^+_P \cup (I_Q \cap o_P)$ and with a similar reasoning processus than for the previous case, we easily deduce that $i'^-_P = i^-_P$. Thus, according to well-formedness of labels, we deduce that $i'_P = i_P$ and $o'_P = o_P$. Moreover, $((s'_P, s_Q), s'_P) \in \mathcal{R}^1_{Sim}$.

   (c) Let us assume we applied P_3 to build the transition: $(s_P, s_Q) \xrightarrow{i'/o'} (s_P, s'_Q)$.Then, there is a transition $s_Q \xrightarrow{i_Q/o_Q} s'_Q \in T_Q$ such that $i'/o' = \Phi_{I_P}(i_Q/o_Q)$. But, $i'_P/o'_P \neq undef$ means that $i^+_Q - o_Q \subseteq I_P$. On the other hand, P and Q are input consistent thus for each trigger i_P in T_P we have $(i^+_Q - i^+_P) \cap I_P = \emptyset$. That also means we have $(i^+_Q - o_P - i^+_P) \cap I_P = \emptyset$ and hence we don't have $i^+_Q - o_Q \subseteq I_P$. Then, the result is that $i'_P/o'_P = undef$ when rule P_3 is applied.

□

To set the main result of this section, we extend the notion of temporal logic property verification to the BDL language. Let ϕ be a $\forall CTL*$ formula and P be a BDL program, we say that $P \models \phi$ if

and only if $\mathcal{K}(\mathcal{S}(P)) \models \phi$. Now, we can prove that BDL composition operators preserve $\forall CTL*$ formulas.

Theorem 2 *Let P and Q be two behavioral programs with disjoint output sets. Let ϕ be a $\forall CTL*$ formula with atomic propositions in the alphabet of P. The following properties hold:*

1. *if $P \models \phi$ then $P[Q/s] \models \phi$*
2. *if P and Q are both input and output consistent then if $P \models \phi$ then $P \| Q \models \phi$*

Proof

Obvious according to proposition 1 and proposition 2. □

4.4 Modular Verification

In the previous section, we have shown that $\forall CTL*$ formulas are preserved through BDL composition operators. Proofs decomposition is an appealing way to ensure safety properties holding and we aim at verifying that some "assume-guarantee" proceeding is available in the proof of properties related to BDL programs.
A method to get modular verification is to use the natural decomposition of the system. Many finite state systems are composed of sub-systems running in parallel and the global specification of such systems can often be decomposed into properties that characterize the behavior of sub-systems. If we can prove that the system satisfies each local property, and if we know that the conjunction of all the local properties implies the overall specification, then we can conclude that the complete system satisfies its specification.
The assume-guarantee paradigm is a powerful mechanism for decomposing a verification task about a system into subtasks about the individual components of the system. The key to assume-guarantee reasoning is to consider each component not in isolation, but in conjunction with assumptions about its context. Suppose that there are two processes M and M'. The behavior of process M depends on the one of process M', the user specifies a set of assumptions that must be satisfied by M' in order to guarantee the correctness of M. Conversely, the behavior of M' also depends of the one of M, the user specifies a set of assumptions that must be satisfied by M in order to guarantee the correctness of M'. By combining the set of assumed and guaranteed properties of M and M' in an appropriate manner, it is possible to deduce the correctness of the system $M \| M'$ without building the global model of the system.
Typically, a formula is a triple $< \phi > M < \psi >$ where ϕ and ψ are temporal logic formulas. The assume-guarantee mechanism is expressed by the following inference rule:

$$\frac{\begin{array}{l} < true > M < \phi > \\ < \phi > M' < \psi > \end{array}}{< true > M \| M' < \psi >}$$

In [30], it is shown that the model of Kripke structures allows to automate this type of reasoning. The preorder $\preceq_K$ over Kripke structures ensures that if $M \models \Phi$ and $M' \preceq_K M$ then $M' \models \phi$, for

$\phi \in \forall CTL*$. Moreover, for each $\forall CTL*$ formula ϕ, a specific Kripke structure: the *tableau* of ϕ denoted by $\mathcal{T}(\phi)$ is defined and $M \models \phi$ if and only if $M \preceq_K \mathcal{T}(\phi)$.
As a consequence, in the Kripke structure context, we can use the assume-guarantee mechanism. Assumptions and specifications are either Kripke structures or $\forall CTL*$ formulas. For instance, let M and M' be two finite state models and A a Kripke structure representing a set of assumptions, then the following proof scheme holds:

$$\frac{\begin{array}{c} M \preceq_K A \\ A \| M' \models \phi \\ \mathcal{T}(\phi) \| M \models < \psi > \end{array}}{M \| M' < \psi >}$$

To extend the assume-guarantee mechanism to BDL programs we have shown that the $\preceq_E$ preorder implies the $\preceq_K$ preorder. The following property shows that the parallel composition over BDL programs is equivalent to composition of the Kripke structures associated with these programs. This is fundamental to ensure that assume-guarantee mechanism can be applied in BDL language in order to get modular model-checking for BDL programs.

Proposition 3 *Let P_1 and P_2 be two behavioral both input and output consistent, $\mathcal{K}(\mathcal{S}(P_1 \| P_2))$ is isomorphic to $\mathcal{K}(\mathcal{S}(P_1)) \|_K \mathcal{K}(\mathcal{S}(P_2))$.*

Proof

First of all, we introduce two notations: Let $S_{P_1 \| P_2}$ be the set of states of $\mathcal{K}(\mathcal{S}(P_1 \| P_2))$ and $R_{P_1 \| P_2}$ its transition relation. We will denote $S_\|$ the set of states of $\mathcal{K}(\mathcal{S}(P_1)) \|_K \mathcal{K}(\mathcal{S}(P_2))$ and $R_\|$ its transition relation.

1. We prove that there is an isomorphism from $S_{P_1 \| P_2}$ to $S_\|$. Let $\mathcal{C}$ be a mapping from $S_{P_1 \| P_2}$ to $S_\|$. We define $\mathcal{C}((s_1, s_2), v) = ((s_1, v \cap A_1), (s_2, v \cap A_2))$. We show that $\mathcal{C}$ is a bijection. Let us assume that $\mathcal{C}((s_1, s_2), v) = \mathcal{C}((t_1, t_2), u)$, then by construction $s_1 = t_1$ and $s_2 = t_2$. Moreover, there are 2 transitions: $(s_1, s_2) \xrightarrow{i/o} (s'_1, s'_2)$ and $(t_1, t_2) \xrightarrow{i'/o'} (t'_1, t'_2)$ in $\mathcal{S}(P_1 \| P_2)$ such that $(i^+ \cup o) \cap A_1 = (i'^+ \cup o') \cap A_1$ and $(i^+ \cup o) \cap A_2 = (i'^+ \cup o') \cap A_2$. According to the well-formedness of LFSM labels, we deduce that $i = i'$ and $o = o'$. Thus, $\mathcal{C}$ is injective.

   On the opposite side, we show that $\mathcal{C}$ is surjective. First of all, we remark that $\mathcal{C}((s_1, s_2), \emptyset) = ((s_1, \emptyset), (s_2, \emptyset)) \in S_\|$ according to Kripke structure construction. Let $((s_1, v_1), (s_2, v_2))$ be a state in $S_\|$. By definition of the composition of Kripke structures, we have: $v_1 \cap A_2 = v_2 \cap A_1$. Moreover, assuming neither $v_1 = \emptyset$ nor $v_2 = \emptyset$, thus $v_1 = i_1^+ \cup o_1$ for some label i_1/o_1 in $\mathcal{S}(P_1)$ and $v_2 = i_2^+ \cup o_2$ for some label i_2/o_2 in $\mathcal{S}(P_2)$ such that there are transitions: $s_1 \xrightarrow{i_1/o_1} s'_1$ in $\mathcal{S}(P_1)$ and $s_2 \xrightarrow{i_2/o_2} s'_2$ in $\mathcal{S}(P_2)$. According to rule P_1 of semantics $\mathcal{S}$, we build a transition $(s_1, s_2) \xrightarrow{(i_1/o_1) \oplus (i_2/o_2)} (s'_1, s'_2)$ in the transition relation of $\mathcal{S}(P_1 \| P_2)$. Then, by construction, there is a state $((s_1, s_2), v)$ in the corresponding Kripke structure and $v = i_1^+ \cup i_2^+ \cup o_1 \cup o_2$. We prove that $v \cap A_1 = v_1$: $v \cap A_1 = (i_1^+ \cup i_2^+ \cup o_1 \cup o_2) \cap A_1$

$= (i_1^+ \cup o_1) \cup ((i_2^+ \cup o_2) \cap A_1$. But, P_1 and P_2 are input and output consistent, so there are not elements of $(i_2^+ \cup o_2) \cap A_1$ not in $i_1^+ \cup o_1$ and so $v \cap A_1 = v_1$. We apply a symmetric reasoning process to prove that $v \cap A_2 = v_2$. Now, we assume that $v_1 = \emptyset$. So by definition of the composition operation for Kripke structures, we have $v_2 \cap A_1 = v_1 \cap A_2 = \emptyset$. Then, there is a transition $s_2 \xrightarrow{i_2/o_2} s_2'$ in $\mathcal{S}(P_2)$ where $v_2 = i_2^+ \cup o_2$. According to rule P_3 of the semantics $\mathcal{S}$, we build a transition: $(s_1, s_2) \xrightarrow{\Phi_{I_1}(i_2/o_2)} (s_1, s_2')$ in transition relation of $\mathcal{S}(P_1 \| P_2)$. Then, by construction, there is a state $((s_1, s_2), v$ in the corresponding Kripke structure and $v = (i_2^+ \cup (I_1 \cap o_2) \cup o_2) = (i_2^+ \cup o_2)$. Thus $v \cap A_2 = v_2$. The symmetric case ($v_2 = \emptyset$ and $v_1 \neq \emptyset$) is similar. Finally, when both v_1 and v_2 are empty, we refer to the preliminary remark to get the result. Hence, $\mathcal{C}$ is a bijection.

2. If $((s_0^1, s_0^2), v)$ is an initial state in $S_{P_1 \| P_2}$ then s_0^1 is an initial state in $\mathcal{S}(P_1))$ and by construction $(s_0^1, v \cap A_1)$ is initial in $\mathcal{K}(\mathcal{S}(P_1))$. Symmetrically, $(s_0^2, v \cap A_2)$ is an initial state in $\mathcal{K}(\mathcal{S}(P_2))$. At the opposite, if $\mathcal{C}(((s_0^1, s_0^2), v))$ is an initial state in $S_\|$, then $((s_0^1, s_0^2), v)$ is initial in $S_{P_1 \| P_2}$.

3. Obviously, $A_1 \cup A_2$ is the alphabet (atomic propositions) of both $S_{P_1 \| P_2}$ and $S_\|$.

4. For transition relations, we will prove that:
   $(((s_1, s_2), v), ((t_1, t_2), u) \in R_{P_1 \| P_2} \Longleftrightarrow (\mathcal{C}(((s_1, s_2), v)), \mathcal{C}(((t_1, t_2), u))) \in R_\|$. First, we consider that $s_1 \neq t_1$ and $s_2 \neq t_2$. According to the definition of $R_{P_1 \| P_2}$, we know that there is a transition $(s_1, s_2) \xrightarrow{i/o} (t_1, t_2)$ in the transition relation of $\mathcal{S}(P_1 \| P_2)$ and $v = i^+ \cup o$. Thus, according to the transition relation construction for the parallel operator, there are two transitions: $s_1 \xrightarrow{i_1/o_1} t_1$ in $\mathcal{S}(P_1)$ and $s_2 \xrightarrow{i_2/o_2} t_2$ in $\mathcal{S}(P_2)$. Moreover, from item(1) we know that both $(s_1, v \cap A_1)$ $(t_1, u \cap A_1)$ are states in $\mathcal{K}(\mathcal{S}(P_1))$ and both $(s_2, v \cap A_2)$ $(t_2, u \cap A_2)$ are states in $\mathcal{K}(\mathcal{S}(P_2))$. Thus, $((s_1, i_1^+ \cup o_1), (t_1, u \cap A_1))$ is in $\mathcal{K}(\mathcal{S}(P_1))$, and $((s_2, i_2^+ \cup o_2), (t_2, u \cap A_2))$ is in $\mathcal{K}(\mathcal{S}(P_2))$. But, $i/o = (i_1/o_1 \oplus i_2/o_2)$ since $s_1 \neq t_1$ and $s_2 \neq t_2$. So, $v \cap A_1 = (i_1^+ \cup o_1) \cup ((i_2^+ \cup o_2) \cap A_1$ and in item (1) we have already proved that $v \cap A_1 = i_1^+ \cup o_1$. Similarly, $v \cap A_2 = i_2^+ \cup o_2$. Then, since P_1 and P_2 are both input and output disjoint, $v \cap A_1 = v \cap A_2$ and so, by definition of the composition of Kripke structures, $(((s_1, v \cap A_1), (s_2, v \cap A_2)), ((t_1, u \cap A_1), (t_2, u \cap A_2)))$ in $R_\|$. As a result: $(((s_1, s_2), v), ((t_1, t_2), u) \in R_{P_1 \| P_2} \Longleftrightarrow (\mathcal{C}(((s_1, s_2), v)), \mathcal{C}(((t_1, t_2), u))) \in R_\|$.

   Now, we assume that $s_1 = t_1$. We consider that $(((s_1, s_2), v), ((s_1, t_2), u) \in R_{P_1 \| P_2}$. Then, there is a transition $s_2 \xrightarrow{i_2/o_2} t_2$ in $\mathcal{S}(P_2)$ and we applied rule P_3 to build the transition: $(s_1, s_2) \xrightarrow{\Phi_{I_1}(i_2/o_2)} (s_1, t_2)$ in $\mathcal{S}(P_1 \| P_2)$ and then $i/o = \Phi_{I_1}(i_2/o_2)$. In item(1), we have already shown that $v \cap A_2 = i_2^+ \cup o_2$ according to the definition of Φ. Then, $((s_2, v \cap A_2), (t_2, u \cap A_2))$ belongs to the transition relation of $\mathcal{K}(\mathcal{S}(P_2))$. On the other hand, $u \cap A_1 = \emptyset$. Assume that it is not, so we did not applied P_3 to build the considered transition in $\mathcal{S}(P_1 \| P_2)$ and this would mean that there is a transition $s_1 \xrightarrow{i_1/o_1} s_1$ in $\mathcal{S}(P_1)$ where $i_1^+ = \emptyset$; this is not allowed (triggers are never empty). Thus, $((s_1, \emptyset), (s_1, \emptyset))$ belongs to the transition relation of $\mathcal{K}(\mathcal{S}(P_2))$ and the consequence is $(((s_1, \emptyset), (s_2, v \cap A_2)), ((s_1, \emptyset), (t_2, u \cap A_2)))$ in $R_\|$. But,

we have $\mathcal{C}((s_1, s_2), v) = ((s_1, \emptyset), (s_2, v \cap A_2))$ and $\mathcal{C}((s_1, t_2), u) = ((s_1, \emptyset), (t_2, u \cap A_2))$, hence we get the result.

The symmetric case ($s_2 = t_2$) supports a symmetric prove and the case where both $s_1 = t_1$ and $s_2 = t_2$ is obvious since $\mathcal{C}(((s_1, s_2), \emptyset)) = ((s_1, \emptyset), (s_2, \emptyset))$ and $(((s_1, s_2), \emptyset), ((s_1, s_2), \emptyset))$ in $R_{P_1 \| P_2}$ and $(((s_1, \emptyset), (s_2, \emptyset)), ((s_1, \emptyset), (s_2, \emptyset)))$ is in $R_{\|}$.

□

5 Practical Issues

5.1 Design Rules

We state here the practical design rules that we can draw from our model and that can be applied at the behavioral language level. When a behavioral program P (called the base program in the following) is extended by another behavioral program P' (called the derived program in the following), respecting these rules ensures that we obtain a new deterministic automaton for which behavioral substitutability holds ($P' \preceq P$). These rules correspond to *sufficient* conditions that save us the trouble of a formal proof for each derived program.
To express these practical rules, we shall use the following notations: as before, for a program P, I_P and O_P denote respectively the input and output event sets of P, that is the input and output alphabets of P. For a state s, $\mathcal{I}_s$ (resp. $\mathcal{O}_s$) denotes the preemption trigger set (resp. the preemption action set) of s, that is the set of the triggers (resp. of the actions) of all outgoing transitions of s[7]. We also define the set of triggers and of actions of a program P, $\mathcal{I}_P$ and $\mathcal{O}_P$, as well as the input and output alphabets, I_s and O_s, of a state s:

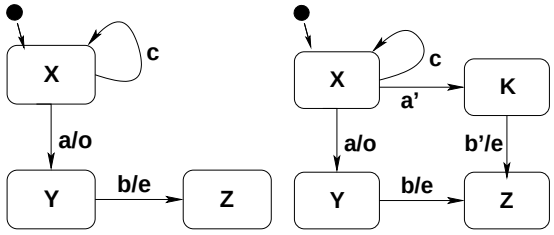
$$\mathcal{I}_P = \bigcup\{\mathcal{I}_s | s \in S_P\} \quad \mathcal{O}_P = \bigcup\{\mathcal{O}_s | s \in S_P\}$$
$$I_s = \bigcup\{t | t \in \mathcal{I}_s\} \quad O_s = \bigcup\{t | t \in \mathcal{O}_s\}$$

where S_P is the set of states of P.
For the time being we have identified eight practical rules, that are listed below. Some of these rules are illustrated with the base program presented on figure 11(a).

1. *No modification of the base program structure*: no state, nor transition from the base program should be deleted; in the same way, target and source state of an existing transition must not be changed;

2. *Addition of independent transitions*: if a new transition t'/a' is added from an existing state s in the base program, then its trigger should not belong to the preemption trigger set of the state ($t' \notin \mathcal{I}_s$); there are no other constraints on the states and transitions reachable through the new t'/a' transition; figure 11(b) shows a correct application of this rule;

[7] Note that, since triggers and actions are themselves sets of events, $\mathcal{I}_s$ and $\mathcal{O}_s$ are sets of subsets of events, whereas I_s, O_s, I_P, and O_P are simply sets of events.



(a) Original (base) program (b) Applying rule 2

Figure 11: On the right, a transition from X to K was added but its trigger a' was not a previous trigger of X; it does not even belong to the base program alphabet

3. *Parallel composition with a program with different actions*: it is allowed to compose a new behavioral program Q in parallel with the base program P if the output set of Q is disjoint from the one of P ($O_P \cap O_Q = \emptyset$); figure 12 shows a violation of this rule;

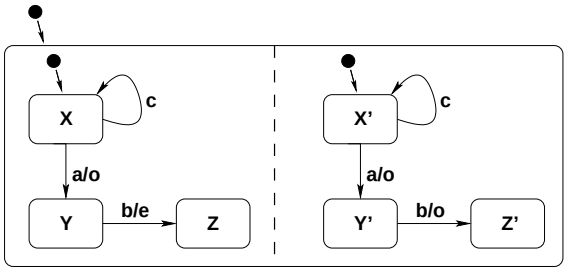
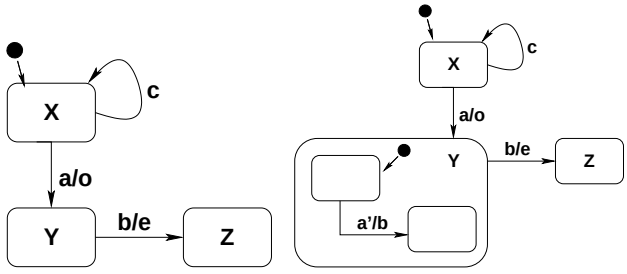


Figure 12: Counter-example for rule 3: the base program (left part) and the program in parallel (right part) share the output event o. The original program has the trace $\{a/o, b/e\}$ whereas the new one has $\{a/o, b/oe\}$

4. *Parallel composition with a program with a different initial trigger*: it is allowed to compose a new behavioral program Q in parallel with the base program P if the preemption trigger sets of the initial states of both P and Q are disjoint ($\mathcal{I}_{s_0^P} \cap \mathcal{I}_{s_0^Q} = \emptyset$);

5. *Parallel composition with a substitutable program*: it is allowed to compose a new behavioral program Q in parallel with the base program P if Q is substitutable for P (that is $Q \preceq P$); this is a consequence of the compositionality property theorems (and of the fact that $P \parallel P$ is P);

6. *Hierarchical composition without auto-preemption*: it is allowed to refine a state s of the base program P with a program Q ($P[Q/s]$) if no action of Q is also a trigger of s ($\mathcal{O}_Q \cap \mathcal{I}_s = \emptyset$); this means that the new nested program cannot terminate the state it refines; figure 13 shows a violation of this rule;



(a) Original (base) program

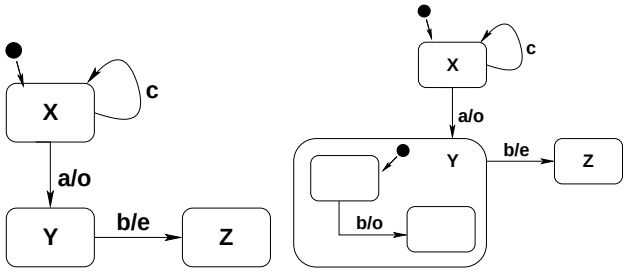
(b) Counter-example for rule 6

Figure 13: In the base program(left part), state Y was waiting for external event b, whereas, in the refinement(right part), b is emitted from the inside

7. *Hierarchical composition with a program with different triggers or actions*: it is allowed to refine a state s of the base program P with a program Q ($P[Q/s]$) if
   (a) either the set of triggers of Q is disjoint from the preemption trigger set of s ($\mathcal{I}_Q \cap \mathcal{I}_s = \emptyset$),
   (b) or the set of triggers of Q intersects the preemption trigger set of s and the output alphabet of Q is disjoint from the output alphabet of s ($O_Q \cap O_s = \emptyset$); this prevents from adding spurious output actions of the base program when preempting both the nested and the enclosing state; figure 14 shows a violation of this rule,

8. *No localization of global events*: a global input or output event (be it a trigger or an action) cannot be made local; thus, local events are acceptable if they do not belong to the base program alphabet.

These design rules are however restrictive and, in some cases, the designer may need to go beyond them to produce an interesting behavior. In this case, we have to apply the *behavioral substitutability algorithm* described in section 3.2 to prove that a derived behavioral program is correct. We are currently implementing this algorithm in a verification tool. As an example, figure 15 shows a case where substitutability holds without applying the design rules.



(a) Original (base) program

(b) Counter-example for rule 7(b)

Figure 14: When state Y is preempted by event b, the action is e in the original program, whereas it is oe in the new program, and with o belonging also to the base alphabet

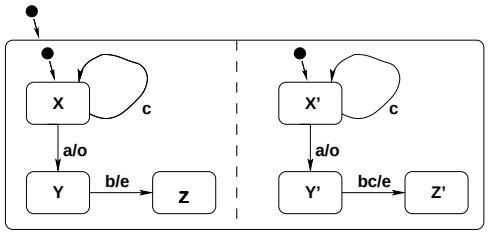


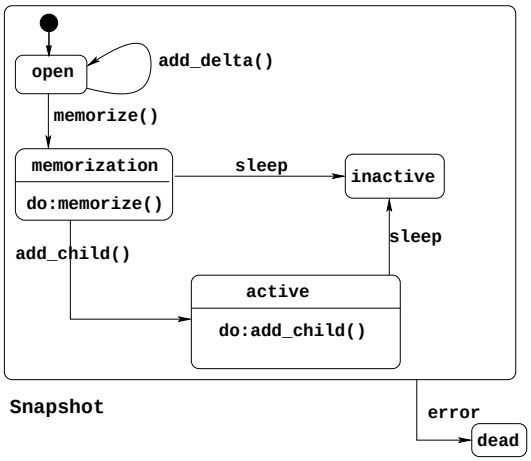
Figure 15: Example of substitutability without applying the design rules: rules 3, 4, and 5 are clearly violated although any trace of the original program (left part) is also a trace of the composed one

5.2 Application to Components

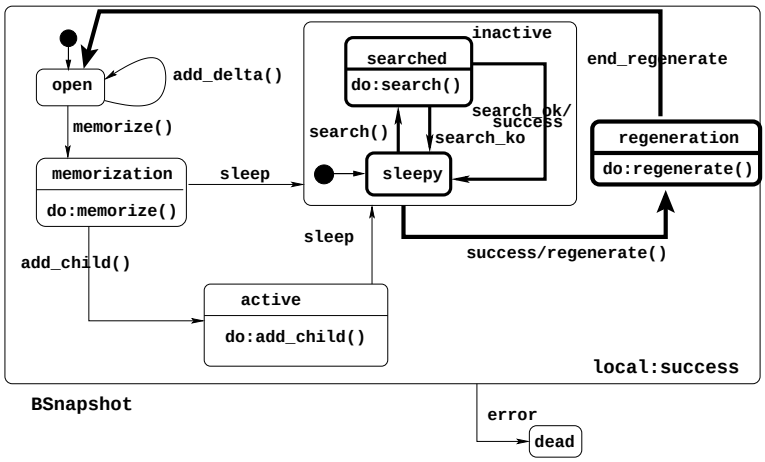
To illustrate our purpose, let us consider the previously mentioned history mechanism (section 2.1). We present on figure 16(a) the behavioral program for the whole `Snapshot` class. This program specifies the valid sequences of operations that can be applied to `Snapshot` instances. Two states correspond to execution of operations (`memorize` and `add_child`); they are to be refined by behavioral programs describing these operations.
Figure 16(b) presents the expected behavioral program for class `BSnapshot` which derives from `Snapshot`. In particular, `BSnapshot` necessitates a new operation, `regenerate`, called when backtracking the history (i.e., when `search` returns `success`). It is clear that the new class sports a behavior significantly different from its base class: it has the extra possibilities to search inside a sleeping snapshot and to call `regenerate` when `success` occurs.
The behavioral program of `BSnapshot` has been obtained from the one of `Snapshot` after applying a combination of our design rules. Obviously no state nor transition have been deleted from `Snapshot` (rule 1). The new transition from `inactive` to `regeneration` bears a completely new trigger (rule 2). The program that refines state `inactive` has no trigger belonging to the preemption trigger set of this state (rule 7(a)). Finally, the local event `success` was not part of the `Snapshot` program (rule 8). Thus, by construction, `BSnapshot` is substitutable for class `Snapshot`; no other verification is necessary to assert that `BSnapshot` $\preceq$ `Snapshot`.
Therefore, even though `BSnapshot` extends the behavior of `Snapshot`, the extension has no influence when a `BSnapshot` is used as a `Snapshot`. As a result, every trace of `Snapshot` is also a trace of `BSnapshot`.

5.3 Stability of Properties

We continue with the previous example. To prove that temporal properties valid for `Snapshot` are still valid for `BSnapshot`, we need to verify that $\mathcal{S}$(`BSnapshot`) $\preceq_E$ $\mathcal{S}$(`Snapshot`) in order to apply the results of section 4. As shown before, the behavorial program of `BSnapshot` has been obtained from the one of `Snapshot` after applying a combination of our design rules (rule 1, rule 2 and rule 7(a)). Note that applying rules 1 or 2 to a program P results in a program P' such that $\mathcal{S}(P') \preceq_E \mathcal{S}(P)$. Indeed, if rule 1 is applied, then we do not modify P ($P = P'$) thus $\mathcal{S}(P')\backslash A_P = \mathcal{S}(P)$. If rule 2 is applied, P' only contains additional transitions whose triggers do not belong to A_P; thus obviously $\mathcal{S}(P')\backslash A_P = \mathcal{S}(P)$ too. Hence, in both cases $\mathcal{S}(P)\ \mathcal{E}_{Sim}\ \mathcal{S}(P')$ and $\mathcal{S}(P') \preceq_E \mathcal{S}(P)$. On the other hand, applying rule 7(a) to a program P also results in a program P' such that $\mathcal{S}(P') \preceq_E \mathcal{S}(P)$ (from proposition 3). Thus it turns out that every temporal property in $\forall CTL^*$ true for `Snapshot` is also true for its extension `BSnapshot`.
For instance, suppose we wish to prove the following property: "It is possible to add a child to a snapshot (i.e., to call the `add_child()` operation) only after memorization has been properly done". Looking at the behavioral program (figure 16(a)), this property (referred to as P_{child}) corresponds to the following behavior. When exiting successfully from state `memorization`, if `add_child()` is received, then control enters state `active`. Then label `sleep` leads to the `inactive` state. Otherwise, operation `memorize()` emits `error` which provokes global preemption. As already mentioned in section 4, it is sufficient to prove that the property holds for the initial state.



(a) Behavioral program of class `Snapshot`.



(b) Behavioral program of class `BSnapshot`. It is similar to `Snapshot` with a refined `inactive` state, a local event `success`, and the possibility of launching `regenerate` from the `inactive` state. Restriction `BSnapshot` $\setminus A$ `Snapshot` is obtained by removing states and transitions displayed with thick lines.

Figure 16: Behavioral programs of classes `Snapshot` and `BSnapshot`.

We are developing a tool that allows us to describe BLOCKS component behavior and to automatically achieve proves of safety properties by calling the *NuSMV* model checker [14]. *NuSMV* suits our needs because it makes it possible to represent synchronous finite state systems and to analyze specifications expressed in $\forall CTL^*$ temporal logic. It uses both symbolic BDD-based and SAT-based (based on propositional satisfiability) model checking techniques. The main novelty is the integration of SAT-based techniques since BDD-based and SAT-based model checking usually solve different classes of problems and therefore can be seen as complementary techniques.
We have decomposed the P_{child} property into two specifications, that are checked in our tool by calling *NuSMV*. The first specification writes:

$$\forall \mathbf{G}(\texttt{add\_child}()\&\forall \mathbf{G}(\neg \texttt{error})) \Rightarrow \forall \mathbf{F}\, \texttt{state} = \texttt{inactive}$$

and the second one:

$$\forall \mathbf{G}(\texttt{error} \Rightarrow \forall \mathbf{G}(\neg \texttt{state} = \texttt{inactive}))$$

Intuitively, the two formulae state that it is always true that

- either the memorization has been correctly executed and a new child has been added; this is what the first specification means: `add_child()` is received, no `error` occurs, and state `inactive` is reached;
- or the memorization failed: `error` occurred, and state `inactive` will never be reached, as expresses by the second specification.

Our tool automatically transforms the description of the behavioral program of `Snapshot` and the two above specifications into acceptable inputs for *NuSMV*. in this case, the tool returns that both specifications are true for `Snapshot`.
Other properties can be checked for `BSnapshot`, for instance, we can prove that

$$\forall [(\texttt{memorize}() \Rightarrow \mathbf{F}(\texttt{state} = \texttt{regeneration}))\ \mathbf{U}\ \texttt{error}]$$

is false for `BSnapshot`. This property expresses that if `memorize()` is called then state `regeneration` is reached provided that `error` never occurs. This is obviously false. In such cases, the diagnosis returned by *NuSMV* is a counter-example. Our tool interprets and displays a user friendly version of this diagnosis for the user.
Although model checkers can prove temporal properties automatically and efficiently, dealing in practice with complex temporal logic formulae is tedious and error-prone. It is more convenient to use the same formalism to express both the component behavior and the temporal property to be proved. This leads to the classical notion of *observers* in model-checking [22]. An observer is a behavioral program expressing a temporal logic property. It is designed so that, once composed in parallel with the program to check, it listens (non intrusively) to the input and output events and emits a failure when the property is violated.
To complete the proof, we could characterize the property by an observer program, then it would be sufficient to consider the behavioral program `Snapshot` $\|$ `Observer` and to verify that no failure is emitted.

The observer technique makes it easier to formulate complicated properties than with temporal logic formulae. In particular, the properties are expressed in the same formalism that is used to represent the component behavior and at the same abstraction level.

6 Discussion and Perspectives

The work described in this paper is derived from our experience and aims at simplifying the correct use of a framework: we discuss this issue in subsection 6.1). To support the corresponding methodology, we are currently considering a series of tools, described in 6.2. Finally, in 6.3, we compare our approach with other work in the area of component compatibility and substitutability.

6.1 Methodological Issues

Framework description

Framework technology is a valuable approach. We have adapted it to the design of knowledge-based system engines and observed a significant gain in development time. For instance, once the analysis completed, the design of a new planning engine [16] only took two months (instead of about two years for a similar former project started from scratch) and more than 90 % of the code reused existing components. Another experiment (for a classification engine) led to similar figures. For both applications we had to extend the BLOCKS framework. While performing these extensions, we realized the need to formalize and verify component protocols, especially when dealing with subtyping. The corresponding formalism, the topic of this paper, has been developed in parallel with the KBS engines. As a consequence of this initial work, developing formal description of BLOCKS components led us to a better organization of the framework, with an architecture that not only satisfies our design rules but also makes the job easier for the framework user to commit to these rules.
Our approach relies on a formal description of the framework architecture as well as of its behavior. Classical UML class diagrams are perfectly suitable for the static description. As far as behavior is concerned, we borrow a StateCharts-like syntax although with a different semantics. The only significant syntactic difference with StateCharts is that we enforce strict encapsulation during refinement (section 3.1). On the other hand, our semantics drastically differs: we use the Synchronous Paradigm [5] because of its capability to propose simple models of automata and efficient formal analysis algorithms. For a more complete discussion about the adequacy of UML to the synchronous paradigm, see [4].

Framework usage

A framework user has to design domain-specific classes and operations based on the entities provided by the framework. These entities come with their own description, both static and behavioral. In our case, the behavioral description uses the language proposed in section 3.1; the same language should allow to describe the additional classes and operations.

To ensure a correct use of the framework, the users may choose among several strategies. First, correctness is enforced, by construction, when using only our operators (section 3.1) and following the design rules (section 5). This is the preferred strategy. Of course our operators are rather primitive and work remains to be done to define libraries and more rules for higher level composition features. Sometimes, this strategy is not applicable. A second (and brute force) possibility is that the framework users set up their own behavior description (still adopting our description formalism); in this case, the system should check the correct behavior of the components *a posteriori*, using a simulator or formal analysis tools.
In addition to this specification-time activities, we can make the behavioral description available at run-time, for instance, by embedding them as assertions into the components code, thus achieving dynamic checking.

6.2 Tools for Correct Framework Manipulation

Our aim is to accompany frameworks with several kinds of dedicated tools.

Tools for behavior description and analysis

We are working on tools for manipulating behavioral programs. Currently, we provide a graphic interface to display existing descriptions and modify them. In the future, the interface will watch the user activity and warn about possible violations of the design rules. Since these rules are just sufficient, it is possible for the user not to apply them or to apply them in such a way that they cannot be clearly identified. To cope with this situation, we shall also provide a static substitutability analyzer, based on our model (section 3.2) and a partitioning algorithm similar to the one in [52].
Another interesting feature would be to provide an automatic code generation facility. Indeed the behavioral description is rather abstract and may be interpreted in a variety of ways. In particular, automata and associated labels can be given a code interpretation. The generated code would provide skeletal implementations of operations. This code will be correct, by construction–at least with respect to those properties which have been previously checked. Furthermore, the generated code can also be instrumented to build run-time traces and assertions in the components.

Tools for property verification

As already mentioned our notion of substitutability guarantees the stability of interesting (safety) properties during the derivation process. Hence, at the user level as well as at the framework one, it may be necessary to automatically verify these properties. To this end, we have chosen model checking techniques instead of theorem proving [2, 46]. Indeed, theorem provers offer a powerful deductive verification mechanism. They can deal with systems of arbitrary size. One major drawback is that they require user interaction to achieve a proof. By contrast, model checkers rely on verification algorithms based on the exploration of a state space. Hence, they can be made automatic since tools are available [34, 26, 27, 44]. They are robust and can be made transparent to framework users.

At the present time we have designed a complete interface with *NuSMV* [14], in both directions. First, our description language is translated into *NuSMV* specifications, and our tool provides also a user friendly way to express the properties the users may want to prove. Second, *NuSMV* diagnosis and return messages are displayed in a readable form: users can browse the hierarchies of behavioral derivations and follow the steps of the proves. It took us a few weeks to connect our behavioral description language to the *NuSMV* model-checker.
The problem with model checkers is the possible explosion of the state space. Fortunately, this problem has become less limiting over the last decade owing to symbolic algorithms. Furthermore, taking advantage of the structural decomposition of the system allows modular proofs on smaller (sub-)systems. This requires a formal model that exhibits the *compositionality property*. As mentioned earlier, this is the case for our model (theorems 1 and 2).

6.3 Related Work

Ensuring correct use of component frameworks through a proof system is a recent research line. Most approaches concentrate on the composition problem [35, 1, 15]. In our case this corresponds to parallel composition.
Modeling component behavior and protocols is an active line of research, especially in the field of Software Architecture [3]. Most works address component compatibility and adaptation in a distributed environment and are often based on process calculi [43, 53, 45]. Some authors put a specific emphasis on the substitutability problem [33]. For instance [9] proposes static subtype checking relying on Nierstrasz's notion of regular types [43]. As another example, in [10], the authors focus on inheritance and extension of behavior, using the π-calculus as their formal model. These works also consider a distributed environment.
The problems of compatibility and substitutability are also significant in fields other than Software Engineering, such as hardware modeling and design. As a matter of example, [12] proposes a "game view" of (hardware) components, relying on deterministic automata.
Our work is close to the one in [12], as far as the objectives (well-formedness, verification, compatibility, and refinement) and models (deterministic automata, non-distributed environment) are concerned although our target applications are similar to the Software Architecture community ones. In contrast with the latter, however, our approach is restricted to the problem of substitutability in a non-distributed world. Indeed this is what we needed for ensuring a safe use of a framework such as BLOCKS. This restriction allows us to adopt models more familiar to software developers (UML StateCharts-like), easier to handle (deterministic systems), efficient for formal analysis (model-checking and simulation), and for which there exist effective algorithms and tools. The Synchronous Paradigm [5] offers good properties and tools in such a context. This is why we could use it as the foundation of our model.

7 Conclusion

The protocol to use frameworks is often complex and the static modeling (*à la* UML) is not sufficient to prevent framework users from fatal misuse. To this end, we assist users by modeling the behavior

of components, thus permitting automatic verification during class derivation and composition. The model has also a pragmatic outcome: it allows simulation of resulting applications and generation of code, of run-time traces, and of run-time assertions.
This behavioral formalism relies on a mathematical model, a specification language, and a semantic mapping from the language to the model. The model supports multiple levels of abstraction, from highly symbolic (just labels) to merely operational (pieces of code). Moreover, this model is original in the sense that it can cover both static and dynamic behavioral properties of components. To use our formalism, the framework user has only to describe behavioral programs, by drawing simple StateCharts-like graphs with a provided graphic interface. The user may be to a large extend oblivious of the theoretical foundations of the underlying models and their complexity.
Presently, we provide a graphical interface for entering behavior descriptions together with a connection to *NuSMV* for automatic property verification and simulation. The next step is to implement the substitutability analysis tool. Then we shall tackle code generation facilities and run-time checks. Developing such tools is a heavy task. Yet, as frameworks are becoming more popular but also more complex, one cannot hope using them without some kind of active assistance, based on formal modeling of component features and automated support.

References

[1] F. Achermann and O. Nierstrasz. Applications = Components + Scripts - A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[2] P. Alexander, M. Rangarajan, and P. Baraona. A Brief Summary of VSPEC. In *World Congress on Formal Methods*, volume 1709 of *LNCS*, pages 1068–1086, 1999.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[4] C. André, M.-A. Peraldi-Frati, and J.-P. Rigault. Integrating the Synchronous Paradigm into UML: Application to Control-Dominated Systems. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *Proceedings of 5th International «UML» Conference*, number 2460 in LNCS, Dresden, Germany, October 2002. Springer-Verlag.

[5] G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stearling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honor of Robin Milner*. MIT Press, 2000.

[6] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem . *Science of Computer Programming*, 24:189–220, 1996.

[7] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson, and M. E. Fayad. Object-Oriented Frameworks: Problems & Experiences. In R. Johnson M. Fayad, D. Schmidt, editor, *Building Application Frameworks: Object Oriented Foundations of Framework Design*. John Wiley, 1999.

[8] J. Bosch, C. Szyperski, and W. Weck. Component-Oriented Programming Workshop. In Jacques Malenfant, Sabine Moisan, and Ana Moreira, editors, *ECOOP'2000 Workshop Reader*, volume 1964 of *LNCS*. Springer, 2000.

[9] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and Tool Support for Debugging Object Protocols. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 50–59, San Diego, CA, USA, 2000. ACM Press.

[10] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, (41):105–138, 2001.

[11] J. Cavarroc, S. Moisan, and J-P. Rigault. Simplifying an Extensible Class Library Interface with OpenC++. In *OOPSLA'98, Worshop on Reflective Programming in C++ and Java*, 1998.

[12] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and Freddy Y. C. Mang. Synchronous and Bidirectional Component Interfaces. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceeeding CAV*, number 2404 in LNCS, pages 214–227, Copenhagen, Danmark, July 2002. Springer-Verlag.

[13] S. Chiba. A Metaobject Protocol for C++. In *OOPSLA'95*, volume 30 of *SIGPLAN Notices*, pages 285–299. ACM Press, 1995.

[14] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: an OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceeeding CAV*, number 2404 in LNCS, pages 359–364, Copenhagen, Danmark, July 2002. Springer-Verlag.

[15] J. Costa Seco and L. Caires. A Basic Model of Typed Components. In Elisa Bertino, editor, *ECOOP 2000*, volume 1850 of *LNCS*, pages 108–128. Springer, 2000.

[16] M. Crubézy. *Pilotage de programmes pour le traitement d'images médicales*. PhD thesis, Université de Nice Sophia Antipolis, 1999.

[17] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2001.

[18] BEA Systems et al. CORBA Component Model Joint Revised Submission, July 1999.

[19] R. Forsyth. *Expert Systems : Principles and Case Studies*. Chapman and Hall, 2nd edition, 1989.

[20] E. Gamma, R. Helm, R. Johson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[21] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[22] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[23] D. Harel and O. Kupferman. On object systems and behavioral inheritance. *IEEE Trans. Software Engineering*, 2002. to appear.

[24] D. Harel and A. Pnueli. On the development of reactive systems. In *NATO, Advanced Study institute on Logics and Models for Verification and Specification of Concurrent Systems*. Springer Verlag, 1985.

[25] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulation on finite and infinite graphs. *Proc. Symp. Foundations of Computer Science, IEEE*, pages 453–462, 1995.

[26] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid System. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[27] G.J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23:279–295, 1997.

[28] R. E. Johnson. Frameworks = (Components + Patterns). *CACM*, 10(40):39–42, 1997.

[29] R.E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[30] E. M. Clarke Jr., O.Grumberg, and D.Peled. *Model Checking*. MIT Press, 2000.

[31] G. Kiczales, J. de Rivière, and D. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[32] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97*, volume 1241. Springer-Verlag, 1997.

[33] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[34] K.L MacMillan. *The SMV Model Checker*. Available from http:www-cad.eecs.berkeley.edu/∼kenmcmil/smv/, 2001.

[35] K. Mani Chandy and M. Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, 20(1):7–21, January 2002.

[36] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Composition. *LNCS: Concur*, 630, 1992.

[37] R. Marvie, P. Merle, and J.M. Geib. Towards a Dynamic CORBA Component Platform. In *2nd International Symposium on Distributed Object Applications (DOA 2000)*, Antwerp, Belgium, September 2000.

[38] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[39] R. Milner. An algebraic definition of simulation between programs. *Proc. Int. Joint Conf. Artificial Intelligence*, pages 481–489, 1971.

[40] S. Moisan. Réutilisation et générateurs de systèmes à base de connaissances : le *framework* BLOCKS. *TSI*, 20(4):529–553, 2001.

[41] S. Moisan, A. Ressouche, and J-P. Rigault. BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems. *Informatica, Special Issue on Component Based Software Development*, 25:501–507, 2001.

[42] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, 3rd edition, October 2001.

[43] Nierstrasz O. *Object-Oriented Software Composition*, chapter Regular Types for Active Objects, pages 99–121. Prentice-Hall, 1995.

[44] P. Pettersson and K. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, 2000.

[45] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), Nov 2002.

[46] M. Rangarajan and P. Alexander. Analysis of Component-Based Systems - An Automated Theorem Proving Approach. In *Specification and Verification of Component-Based Systems (SAVCBS'2001) Workshop at OOPSLA'2001*, October 2001.

[47] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[48] I. Ryl, M. Clerbout, and A. Bailly. A Component Oriented Notation for Behavioral Specification and Validation. In *Specification and Verification of Component-Based Systems (SAVCBS'2001) Workshop at OOPSLA'2001*, October 2001.

[49] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. http://www.ObjecTime.com, 1998.

[50] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[51] C. Szyperski, J. Bosch, and W. Weck. Component-Oriented Programming Workshop. In Ana Moreira and Serge Demeyer, editors, *ECOOP'99 Workshop Reader*, volume 1743 of *LNCS*. Springer, 1999.

[52] Li Tan and R. Cleaveland. Simulation revisited. In Tiziana Margaria and Wang Yi, editors, *Proceedings TACAS 2001*, number 2031 in LNCS, pages 480–495. Springer-Verlag, 2001.

[53] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399